Determining the Coverage of a Test Suite

Richard C. Waters

MIT AI Laboratory
545 Technology Sq.
Cambridge MA 02139
Dick@AI.MIT.EDU

Mitsubishi Electric Research Laboratories
201 Broadway
Cambridge MA 02139
Dick@MERL.COM

The value of a suite of test cases depends critically on its *coverage*. Ideally a suite should test every facet of the specification for a program and every facet of the algorithms used to implement the specification. Unfortunately, there is no practical way to be sure that complete coverage has been achieved. However, something should be done to assess the coverage of a test suite, because a test suite with poor coverage has little value.

A traditional approximate method of assessing the coverage of a test suite is to check that every condition tested by the program is exercised. For every predicate in the program, there should be at least one test case that causes the predicate to be true and one that causes it to be false. Consider the function `my*` in Figure 1, which uses a convoluted algorithm to compute the product of two numbers.

The function `my*` contains two predicates, `(minusp x)` and `(minusp y)`, which lead to four conditions: `x` is negative, `x` is not negative, `y` is negative, and `y` is not negative. To be at all thorough, a test suite must contain tests exercising all four of these conditions. For instance, any test suite that fails to exercise the condition where `y` is negative will fail to detect the bug in the next to last line of the function.

```
(defun my* (x y)
  (let ((sign 1))
    (when (minusp x)
      (setq sign (- sign))
      (setq x (- x)))
    (when (minusp y)
      (setq sign (- sign))
      (setq y (- x)))
    (* sign x y)))
```

Figure 1: An example program.

(As an example of the fact that covering all the conditions in a program does not guarantee that every facet of either the algorithm or the specification will be covered, consider the fact that the two test cases `(my* 2.1 3)` and `(my* -1/2 -1/2)` cover all four conditions. However, they do not detect the bug on the next to last line and they do not detect the fact that `my*` fails to work on complex numbers.)

The COVER system determines which conditions tested by a program are exercised by a given test suite. This is no substitute for thinking hard about the coverage of the test suite. However, it provides a useful starting point and can indicate some areas where additional test cases should be devised.

User's Manual for COVER

The functions, macros, and variables that make up the COVER system are in a package called `"COVER"`. The six exported symbols are documented below.

- `cover:annotate` *t-or-nil*

  Evaluating `(cover:annotate t)` triggers the processing of function and macro definitions by the COVER system. Each subsequent instance of `defun` or `defmacro` is altered by adding annotation that maintains information about the various conditions tested in the body.

  Evaluating `(cover:annotate nil)` stops the

special processing of function and macro definitions. Subsequent definitions are not annotated. However, if a function or macro that is currently annotated is redefined, the new definition is annotated as well.

The macro `cover:annotate` should only be used as a top-level form. When annotation is triggered, a warning message is printed, and `t` is returned. Otherwise, `nil` is returned.

```
(cover:annotate t) ⇒ t ; after printing:
;;; Warning: Coverage annotation applied.
```

- `cover:forget-all`

  This function, which always returns t, has the effect of removing all coverage annotation from every function and macro. It is appropriate to do this before completely recompiling the system being tested or before switching to a different system to be tested.

- `cover:reset`

  Each condition tested by an annotated function and macro is associated with a flag that trips when the condition is exercised. The function `cover:reset` resets all these flags, and returns `t`. It is appropriate to do this before rerunning a test suite to reevaluate its coverage.

- `cover:report &key` *fn out all*

  This function displays the information maintained by COVER, returning no values. *Fn* must be the name of an annotated function or macro. If *fn* is specified, a report is printed showing information about that function or macro only. Otherwise, reports are printed about every annotated function and macro.

  Out, which defaults to `*standard-output*`, must either be an output stream or the name of a file. It specifies where the reports should be printed.

  If *all*, which defaults to `nil`, is non-null then the reports printed contain information about every condition. Otherwise, the reports are abbreviated to highlight key conditions that have not been exercised.

- `cover:*line-limit*` default value `75`

  The output produced by `cover:report` is truncated to ensure that it is no wider than `cover:*line-limit*`.

```
(setq cover:*line-limit* 43) ⇒ 43

(cover:reset) ⇒ T

(cover:report) ⇒ ; after printing:
;- :REACH (DEFUN MY* (X Y))  <1>

(my* 2 2) ⇒ 4

(cover:report) ⇒ ; after printing:
;+ :REACH (DEFUN MY* (X Y))  <1>
; + :REACH (WHEN (MINUSP X) (SETQ S  <2>
;  - :NON-NULL (MINUSP X)  <4>
; + :REACH (WHEN (MINUSP Y) (SETQ S  <6>
;  - :NON-NULL (MINUSP Y)  <8>

(my* -2 2) ⇒ -4

(cover:report) ⇒ ; after printing:
;+ :REACH (DEFUN MY* (X Y))  <1>
; + :REACH (WHEN (MINUSP Y) (SETQ S  <6>
;  - :NON-NULL (MINUSP Y)  <8>

(cover:report :all t) ⇒ ; after printing:
;+ :REACH (DEFUN MY* (X Y))  <1>
; + :REACH (WHEN (MINUSP X) (SETQ S  <2>
;  + :NON-NULL (MINUSP X)  <4>
;  + :NULL (MINUSP X)  <5>
; + :REACH (WHEN (MINUSP Y) (SETQ S  <6>
;  - :NON-NULL (MINUSP Y)  <8>
;  + :NULL (MINUSP Y)  <9>
```

Figure 2: Example COVER reports.

An example. Suppose that the function `my*` in Figure 1 has been annotated and that no other functions or macros have been annotated. Figure 2 illustrates the operation of COVER and the reports printed by `cover:report`.

Each line in a report contains three pieces of information about a point in a definition: +/- specifying that the point either has (+) or has not (-) been exercised, a message indicating the physical and logical placement of the point in the definition, and in angle brackets < >, an integer that is a unique identifier for the point. Indentation is used to indicate that some points are subordinate to others in the sense that the subordinate points cannot be exercised without also exercising their superiors. The order of the lines of the report is the same as the order of the points in the definition.

Each message contains a label (e.g., `:REACH`, `:NULL`) and a piece of code. There is a point labeled `:REACH` corresponding to each definition as

a whole and each conditional form within each definition. Subordinate points corresponding to the conditions a conditional form tests are grouped under the point corresponding to the form. As discussed in detail in the next subsection, the messages for the subordinate points describe the situations in which the conditions are exercised. Lines that would otherwise be too long to fit on one line have their messages truncated (e.g., points <2> and <6> in Figure 2).

The first three reports in Figure 2 are abbreviated based on two principles. First, if a point p and all of its subordinates have been exercised, then p and all of its subordinates are omitted from the report. This is done to focus the user's attention on the points that have not been exercised.

Second, if a point p has not been exercised, then all of the points subordinate to it are omitted from the report. This reflects the fact that it is not possible for any of these subordinate points to have been exercised and one cannot devise a test case that exercises any of the subordinate points without first figuring out how to exercise p.

An additional complicating factor is that COVER operates in an incremental fashion and does not, in general, have full information about the subordinates of points that have not been exercised. As a result, it is not always possible to present a complete report. However, one can have total confidence that if the report says that every point has been exercised, this statement is based on complete information.

The first report in Figure 2 shows that none of the points within `my*` has been exercised. The second report displays most of the points in `my*`, to set the context for the two points that have not been exercised. The third report omits <2> and its subordinates, since they have all been exercised. The fourth report shows a complete report corresponding to the third abbreviated report.

- `cover:forget &rest` *ids*

This function gives the user greater control over the reports produced by `cover:report`. Each *id* must be an integer identifying a point. All information about the specified points (and their subordinates) is forgotten. From the point of view of `cover:report`, the effect is as if the points never existed. (A forgotten point can be retrieved by reevaluating or recompiling the function or macro definition containing it.) The example below, which follows on after the end of Figure 2, shows the action of `cover:forget`.

```
(cover:forget 6) ⇒ T

(cover:report :all t) ⇒ ; after printing:
;+ :REACH (DEFUN MY* (X Y))  <1>
; + :REACH (WHEN (MINUSP X) (SETQ S  <2>
;  + :NON-NULL (MINUSP X)  <4>
;  + :NULL (MINUSP X)  <5>

(cover:report) ⇒ ; after printing
;All points exercised.
```

The abbreviated report above does not describe any points, because every point in `my*` that has not been forgotten has been exercised. It is appropriate to forget a point if there is some reason that no test case can possibly exercise the point. However, it is much better to write your code so that every condition can be tested.

(Point numbers are assigned based on the order in which points are entered into COVER's database. In general, whenever a definition is reevaluated or recompiled, the numbers of the points within it change.)

The way conditionals are annotated. Figure 3 shows a file that makes use of COVER. Figure 4 shows the kind of report that might be produced by loading the file. Because, `maybe-` and `g` are the only definitions that have been annotated, these are the only definitions that are reported on. The order of the reports is the same as the order in which the definitions were compiled. The report on `g` indicates that the tests performed by `run-tests` exercise most of the conditions tested by `g`. However, they do not exercise the situation in which the `case` statement is reached, but neither of its clauses is selected.

There are no points within `maybe-`, because the code for `maybe-` does not contain any conditional forms. It is interesting to consider the precise points that COVER includes for `g`.

```
(in-package "USER")

(require "COVER" ...)

(defmacro maybe+ (x y)
  '(if (numberp ,x) (+ ,x ,y)))

(cover:annotate t)

(defmacro maybe- (x y)
  '(if (numberp ,x) (- ,x ,y)))

(defun g (x y)
   (cond ((and (null x) y) y)
         (y (case y
              (1 (maybe- x y))
              (2 (maybe+ x y))))))

(cover:annotate nil)

(defun h (x y) ...)

(cover:reset)

(run-tests)

(cover:report :out "report" :all t)
```

Figure 3: Example of a file using COVER.

When COVER processes a definition, a cluster of points is generated corresponding to each conditional form (i.e., **if**, **when**, **until**, **cond**, **case**, **typecase**, **and**, and **or**) that is literally present in the program. In addition, points are generated corresponding to conditional forms that are produced by macros that are annotated (e.g., the **if** produced by the **maybe-** in the first **case** clause in **g**). However, annotation is not applied to conditionals that come from other sources (e.g., from macros that are defined outside of the system being tested). These conditionals are omitted, because there is no reasonable way for the user to know how they relate to the code, and therefore there is no reasonable way for the user to devise a test case that will exercise them.

The messages associated with a point's subordinates describe the situations under which the subordinates are exercised. The pattern of messages associated with **case** and **typecase** is illustrated by the portion (reproduced below) of Figure 4 that describes the **case** in **g**.

```
;    + :REACH (CASE Y (1 (MAYBE- X Y  <13>
;     + :SELECT 1  <15>
;     + :SELECT 2  <16>
;     - :SELECT-NONE  <17>
```

```
;+ :REACH (DEFMACRO MAYBE- (X Y))  <1>
;+ :REACH (DEFUN G (X Y))  <2>
; + :REACH (COND ((AND # Y) Y) (Y (  <3>
;  + :REACH (AND (NULL X) Y)  <9>
;   + :FIRST-NULL (NULL X)  <11>
;   + :EVAL-ALL Y  <12>
;  + :FIRST-NON-NULL (AND (NULL X)   <5>
;  + :FIRST-NON-NULL Y  <7>
;   + :REACH (CASE Y (1 (MAYBE- X Y  <13>
;    + :SELECT 1  <15>
;     + :REACH (IF (NUMBERP X) (- X  <18>
;      + :NON-NULL (NUMBERP X)  <20>
;      + :NULL (NUMBERP X)  <21>
;    + :SELECT 2  <16>
;    - :SELECT-NONE  <17>
;  + :ALL-NULL  <8>
```

Figure 4: The report created by Figure 3.

There are two subpoints corresponding to the two clauses of the **case**. In addition, since the last clause does not begin with **t** or **otherwise**, there is an additional point corresponding to the situation where none of the clauses of the **case** are executed.

The pattern of messages associated with a **cond** is illustrated by the portion (reproduced below) of Figure 4 that describes the **cond** in **g**.

```
; + :REACH (COND ((AND # Y) Y) (Y (  <3>
;  + :REACH (AND (NULL X) Y)  <9>
;  + :FIRST-NON-NULL (AND (NULL X)   <5>
;  + :FIRST-NON-NULL Y  <7>
;  + :ALL-NULL  <8>
```

There are subordinate points corresponding to the two clauses and the situation where neither clause is executed. There is also a point <9> corresponding to the **and** that is the predicate of the first **cond** clause. This point is placed directly under <3>, because it is not subordinate to any of the individual **cond** clauses.

The treatment of **and** (and **or**) is particularly interesting. Sometimes **and** is used as a control construct on a par with **cond**. In that situation, it is clear that **and** should be treated analogously to **cond**. However, at other times, **and** is used to compute a value that is tested by another conditional form. In that situation, COVER could choose to treat **and** as a simple function. However, it is nevertheless still reasonable to think of an **and** as having conditional points that correspond to different reasons why

the and returns a true or false value. It is wise to include tests corresponding to each of these different reasons.

The pattern of messages associated with an and is illustrated by the portion (reproduced below) of Figure 4 that describes the and in g.

```
(cover:report :all t)
;   + :REACH (AND (NULL X) Y)  <9>
;    + :FIRST-NULL (NULL X)  <11>
;    + :EVAL-ALL Y  <12>
```

The final subpoint corresponds to the situation where all of the arguments of the and have been evaluated. The and then returns whatever the final argument returned.

Figure 3 illustrates a batch-oriented use of COVER. However, COVER is most effectively used in an interactive way. It is recommended that you first create as comprehensive a test suite as you can and capture it using a tool such as RT [1]. The tests should then be run in conjunction with COVER and repeated reports from COVER generated as additional tests are created until complete coverage of conditions has been achieved. To robustly support this mode of operation, COVER has been carefully designed so that it will work with batch-compiled definitions, incrementally-compiled definitions, and interpreted definitions.

How COVER Works

The code for COVER is shown in Figures 5, 7, 8, and 10. Figure 5 shows the definition of the primary data structure used by COVER and some of the central operations. A point structure contains five pieces of information about a position in the code for a definition.

hit	Flag indicating whether the point has been exercised.
id	Unique integer identifier.
status	Flag that controls reporting.
name	Logical name.
subs	List of subordinate points.

The hit flag operates as a "time stamp". When a point is exercised, this is recorded by storing the current value of the variable *hit* in the hit field of the point. This method of operation makes it possible to reset the hit flags of all the points currently in existence without visiting any of them (see the definition of cover:reset).

The id is printed in reports and used to identify points when calling cover:forget. The variable *count* is used to generate the values.

The status controls the reporting of a point. It is either :SHOW (shown in reports), :HIDDEN (not shown in reports, but its subordinates may be), or :FORGOTTEN (neither it nor its subordinates are shown in reports). (cover:forget changes the status of the indicated points to :FORGOTTEN.)

The name of a point p describes its position in the definition containing it. A name has the form: (*label code . superior-name*) where *label* is an explanatory label such as :REACH or :NULL, *code* is a piece of code, and *superior-name* is the name of the point containing p (if any). Taken together, the *label* and *code* indicate the position of p in a definition and the condition under which it is exercised (see the discussion of Figure 4).

At any given moment, the variable *points* contains a list of points corresponding to the annotated definitions known to COVER. (The function cover:forget-all resets *points* to nil.) As an illustration of the point data structure, Figure 6 shows the contents of *points* corresponding to the second report in Figure 2. It is assumed that *hit* has the value 1.

The function add-top-point adds a new top-level point corresponding to a definition to the list *points*. If there is already a point for the definition, the new point is put in the same place in the list.

The function record-hit records the fact that a point has been exercised. This may require locating the point in *points* using locate or adding the point into *points* using add-point. record-hit is optimized so that it is extremely fast when the point has already been exercised. This allows COVER to run with relatively little overhead. (The details of the way record-hit and add-point operate are discussed further in conjunction with Figure 10.)

```
(in-package "COVER" :use '("LISP"))

(provide "COVER")

(shadow '(defun defmacro))

(export '(annotate report reset forget
          forget-all *line-limit*))

(defstruct (point (:conc-name nil)
                  (:type list))
  (hit 0)
  (id nil)
  (status :show)
  (name nil)
  (subs nil))

(defvar *count* 0)
(defvar *hit* 1)
(defvar *points* nil)
(defvar *annotating* nil)
(defvar *testing* nil)

(lisp:defun forget (&rest ids)
  (forget1 ids *points*)
  t)

(lisp:defun forget1 (names ps)
  (dolist (p ps)
    (when (member (id p) names)
      (setf (status p) :forgotten))
    (forget1 names (subs p))))

(lisp:defun forget-all ()
  (setq *points* nil)
  (setq *hit* 1)
  (setq *count* 0)
  t)

(lisp:defun reset () (incf *hit*) t)
```

```
(lisp:defun add-top-point (p)
  (setq p (copy-tree p))
  (let ((old (find (fn-name p) *points*
                   :key #'fn-name)))
    (cond (old (setf (id p) (id old))
               (nsubstitute p old *points*))
          (t (setf (id p) (incf *count*))
             (setq *points*
                   (nconc *points*
                          (list p)))))))

(lisp:defun record-hit (p)
  (unless (= (hit p) *hit*)
    (setf (hit p) *hit*)
    (let ((old (locate (name p))))
      (if old
          (setf (hit old) *hit*)
          (add-point p)))))

(lisp:defun locate (name)
  (find name
        (if (not (cdr name))
            *points*
            (let ((p (locate (cdr name))))
              (if p (subs p))))
        :key #'name :test #'equal))

(lisp:defun add-point (p)
  (let ((sup (locate (cdr (name p)))))
    (when sup
      (setq p (copy-tree p))
      (setf (subs sup)
            (nconc (subs sup) (list p)))
      (setf (id p) (incf *count*))
      (dolist (p (subs p))
        (setf (id p) (incf *count*))))))
```

Figure 5: The basic data structure used by COVER.

```
((1 :SHOW 1 (#1=(:REACH (DEFUN MY* (X Y))))
  ((2 :SHOW 1 (#2=(:REACH (WHEN (MINUSP X) (SETQ SIGN (- SIGN)) (SETQ X (- X)))) #1#)
    ((3 :HIDDEN 1 ((:REACH (MINUSP X)) #2# #1#) NIL)
     (4 :SHOW 0 ((:NON-NULL (MINUSP X)) #2# #1#) NIL)
     (5 :SHOW 1 ((:NULL (MINUSP X)) #2# #1#) NIL)))
   (6 :SHOW 1 (#6=(:REACH (WHEN (MINUSP Y) (SETQ SIGN (- SIGN)) (SETQ Y (- X)))) #1#)
    ((7 :HIDDEN 1 ((:REACH (MINUSP Y)) #6# #1#) NIL)
     (8 :SHOW 0 ((:NON-NULL (MINUSP Y)) #6# #1#) NIL)
     (9 :SHOW 1 ((:NULL (MINUSP Y)) #6# #1#) NIL))))))
```

Figure 6: The contents of `*points*` corresponding to the second report in Figure 2.

Figure 7 shows the code that prints reports. As can be seen by a comparison of Figures 2 and 6, reports are a relatively straightforward printout of parts of `*points*` with nesting indicated by indentation and only the first part of each point's name shown. The function `report2` supports the abbreviation described in conjunction with Figure 2.

Annotating definitions. Figure 8 shows the code that controls the annotation of definitions by COVER. The first time `cover:annotate` is called, it uses `shadowing-import` to install new definitions for `defun` and `defmacro`. Whether or not annotation is in effect is recorded in the variable `*annotate*`. The variable `*testing*` is used to make it easier to test COVER using

```
(defvar *line-limit* 75)

(proclaim '(special *depth* *all*
                    *out* *done*))

(lisp:defun report
            (&key (fn nil)
                  (out *standard-output*)
                  (all nil))
 (let (p)
  (cond
   ((not (streamp out))
    (with-open-file
        (s out :direction :output)
      (report :fn fn :all all :out s)))
    ((null *points*)
     (format out
       "~%No definitions annotated."))
    ((not fn)
     (report1 *points* all out))
    ((setq p (find fn *points*
                   :key #'fn-name))
     (report1 (list p) all out))
    (t (format out "~%~A is not annotated."
               fn))))
  (values))

(lisp:defun fn-name (p)
  (let ((form (cadr (car (name p)))))
    (and (consp form)
         (consp (cdr form))
         (cadr form))))

(lisp:defun report1 (ps *all* *out*)
  (let ((*depth* 0) (*done* t))
    (mapc #'report2 ps)
    (when *done*
      (format *out*
        "~%;All points exercised."))))

(lisp:defun report2 (p)
  (case (status p)
    (:forgotten nil)
    (:hidden (mapc #'report2 (subs p)))
    (:show
     (cond ((reportable-subs p)
            (report3 p)
            (let ((*depth* (1+ *depth*)))
              (mapc #'report2 (subs p))))
           ((reportable p)
            (report3 p))))))

(lisp:defun reportable (p)
  (and (eq (status p) :show)
       (or *all*
           (not (= (hit p) *hit*)))))

(lisp:defun reportable-subs (p)
  (and (not (eq (status p) :forgotten))
       (or *all* (not (reportable p)))
       (some #'(lambda (s)
                 (or (reportable s)
                     (reportable-subs s)))
             (subs p))))

(lisp:defun report3 (p)
  (setq *done* nil)
  (let* ((*print-pretty* nil)
         (*print-level* 3)
         (*print-length* nil)
         (m (format nil
                    "~;~V@T~:[-~;+~]~{ ~S~}"
                    *depth*
                    (= (hit p) *hit*)
                    (car (name p))))
         (limit (- *line-limit* 8)))
    (when (> (length m) limit)
      (setq m (subseq m 0 limit)))
    (format *out* "~%~A  <~S>" m (id p))))
```

Figure 7: The code for the part of COVER that prints reports.

RT [1].

Redefining `defun` and `defmacro` is a convenient approach to use for supporting COVER, however, it is in general a rather dangerous thing to do. One problem is that for COVER to operate correctly, `cover:annotate` must be executed before any of the definitions you wish to annotate are read. For instance, Figure 3 would not work if an `eval-when` were wrapped around the top-level forms as a group.

When annotation is in effect, the new definitions of `defun` and `defmacro` use `sublis` to replace every instance of `if`, `cond`, etc. with special macros `c-if`, `c-cond`, etc. (see Figure 10). Defining forms created by the user (e.g., `def` in Figure 10) are typically macros that expand into `defmacro`. They are indirectly supported by COVER, as long as their definitions are read after `cover:annotate` has been evaluated.

On the face of it, it is not correct to use `sublis` to rename forms in code, because every instance of the indicated symbols is changed, whether or not they are actually uses of the indicated forms and whether or not they are in quoted lists. Nevertheless, COVER uses `sublis` for two reasons.

First, in contrast to a code walker, `sublis` is very simple. (The only understanding of Lisp structure that COVER needs is how to separate the declarations from the body of a definition, see the function `parse-body`.)

Most problems can easy be avoid by resist-

```
(lisp:defmacro annotate (t-or-nil)
  '(eval-when (eval load compile)
     (annotate1 ,t-or-nil)))

(lisp:defun annotate1 (flag)
  (shadowing-import
   (set-difference '(defun defmacro)
     (package-shadowing-symbols *package*)))
  (when (and flag (not *testing*))
    (warn "Coverage annotation applied."))
  (setq *annotating* (not (null flag))))

(lisp:defmacro defun (n argl &body b)
  (process 'defun 'lisp:defun n argl b))

(lisp:defmacro defmacro (n a &body b)
  (process 'defmacro 'lisp:defmacro n a b))

(lisp:defun parse-body (body)
  (let ((decls nil))
    (when (stringp (car body))
      (push (pop body) decls))
    (loop (unless (and (consp (car body))
                       (eq (caar body)
                           'declare))
            (return nil))
          (push (pop body) decls))
    (values (nreverse decls) body)))
```

```
(defvar *check*
  '((or . c-or) (and . c-and)
    (if . c-if) (when . c-when)
    (unless . c-unless)
    (cond . c-cond) (case . c-case)
    (typecase . c-typecase)))

(lisp:defun process (cdef def fn argl b)
  (if (not (or *annotating*
               (find fn
                     *points*
                     :key #'fn-name)))
      '(,def ,fn ,argl ., b)
      (multiple-value-bind (decls b)
          (parse-body b)
        (setq b (sublis *check* b))
        (let ((name
               '((:reach
                  (,cdef ,fn ,argl)))))
          '(eval-when (eval load compile)
            (add-top-point
              ',(make-point :name name))
            (,def ,fn ,argl ,@ decls
                  ,(c0 (make-point :name
                                   name)
                       name b)))))))
```

Figure 8: The code for the part of COVER that annotates definitions.

```
(EVAL-WHEN (EVAL LOAD COMPILE)
  (COVER::ADD-TOP-POINT '(NIL :SHOW 0 (#1=(:REACH (DEFUN MY* (X Y)))) NIL))
(LISP:DEFUN MY* (X Y)
  (COVER::RECORD-HIT '(NIL :SHOW 0 (#1#) NIL))
  (LET ((SIGN 1))
    (COVER::RECORD-HIT
     '(NIL :SHOW 0 (#2=(:REACH (WHEN (MINUSP X) (SETQ SIGN (- SIGN)) (SETQ X (- X)))) #1#)
       ((NIL :HIDDEN 0 ((:REACH (MINUSP X)) #2# #1#) NIL)
        (NIL :SHOW 0 ((:NON-NULL (MINUSP X)) #2# #1#) NIL)
        (NIL :SHOW 0 ((:NULL (MINUSP X)) #2# #1#) NIL))))
    (IF (PROGN (COVER::RECORD-HIT '(NIL :HIDDEN 0 ((:REACH (MINUSP X)) #2# #1#) NIL))
               (MINUSP X))
        (PROGN (COVER::RECORD-HIT '(NIL :SHOW 0 ((:NON-NULL (MINUSP X)) #2# #1#) NIL))
               (SETQ SIGN (- SIGN)) (SETQ X (- X)))
        (PROGN (COVER::RECORD-HIT '(NIL :SHOW 0 ((:NULL (MINUSP X)) #2# #1#) NIL))
               NIL))
    ...)))
```

Figure 9: Part of the annotated definition of `my*` from Figure 1.

ing the temptation to use `if`, `cond`, etc. as variable names. Any remaining difficulties can be tolerated because COVER is merely part of scaffolding for testing a system rather than part of the system to be delivered. A subtle difficulty concerns `and` and `or`. They are used as type specifiers as well as conditional forms. This difficulty is partly overcome by the type definitions at the end of Figure 10.

Second, the use of `sublis` supports two key features of COVER that would be very difficult to support using a code walker. It insures that only conditional forms that literally appear in the definition are annotated (as opposed to ones that come from macro expansions), and yet, conditionals that come from the expansion of annotated macros are annotated. (Note that the literals that turn into conditionals in the

```
(defvar *fix*
  '((c-or . or) (c-and . and) (c-if . if)
    (c-when . when) (c-unless . unless)
    (c-cond . cond) (c-case . case)
    (c-typecase . typecase)))

(proclaim '(special *subs* *sup*))

(lisp:defmacro sup-mac () nil)

(lisp:defmacro def (name args form)
  `(lisp:defmacro ,name (&whole w ,@ args
                         &environment env)
     (let* ((*subs* nil)
            (*sup*
             `((:reach ,(sublis *fix* w))
               .,(macroexpand-1
                  (list 'sup-mac) env)))
            (p (make-point :name *sup*))
            (form ,form))
       (setf (subs p) (nreverse *subs*))
       (c0 p *sup* (list form)))))

(lisp:defmacro c (body &rest msg)
  (c1 `(list ,body) msg :show))

(lisp:defmacro c-hide (b)
  (c1 `(list ,b) (list :reach b) :hidden))

(eval-when (eval load compile)

(lisp:defun c1 (b m s)
  `(let ((n (cons (sublis *fix*
                          (list .,m))
                  *sup*)))
     (push (make-point :name n :status ,s)
           *subs*)
     (c0 (make-point :name n :status ,s)
         n ,b)))

(lisp:defun c0 (p sup b)
  `(macrolet ((sup-mac () ',sup))
     (record-hit ',p)
     .,b)) )

(def c-case (key &rest cs)
  `(case ,(c-hide key)
     .,(c-case0 cs)))

(def c-typecase (key &rest cs)
  `(typecase ,(c-hide key)
     .,(c-case0 cs)))

(lisp:defun c-case0 (cs)
  (let ((stuff (mapcar #'c-case1 cs)))
    (when (not (member (caar (last cs))
                       '(t otherwise)))
      (setq stuff
        (nconc stuff
          `((t ,(c nil :select-none))))))
    stuff))

(lisp:defun c-case1 (clause)
  `(,(car clause)
    ,(c `(progn ., (cdr clause)) :select
        (car clause))))

(def c-if (pred then &optional (else nil))
  `(if ,(c-hide pred)
       ,(c then :non-null pred)
       ,(c else :null pred)))

(def c-when (pred &rest actions)
  `(if ,(c-hide pred)
       ,(c `(progn ., actions)
           :non-null pred)
       ,(c nil :null pred)))

(def c-unless (pred &rest actions)
  `(if (not ,(c-hide pred))
       ,(c `(progn ., actions) :null pred)
       ,(c nil :non-null pred)))

(def c-cond (&rest cs)
  (c-cond0 (gensym) cs))

(lisp:defun c-cond0 (var cs)
  (cond ((null cs) (c nil :all-null))
        ((eq (caar cs) t)
         (c (if (cdar cs)
                `(progn .,(cdar cs))
                t)
            :first-non-null t))
        ((cdar cs)
         `(if ,(c-hide (caar cs))
              ,(c `(progn .,(cdar cs))
                  :first-non-null
                  (caar cs))
              ,(c-cond0 var (cdr cs))))
        (t `(let ((,var
                   ,(c-hide (caar cs))))
              (if ,var
                  ,(c var :first-non-null
                      (caar cs))
                  ,(c-cond0 var
                            (cdr cs)))))))

(def c-or (&rest ps) (c-or0 ps))

(lisp:defun c-or0 (ps)
  (if (null (cdr ps))
      (c (car ps) :eval-all (car ps))
      (let ((var (gensym)))
        `(let ((,var ,(c-hide (car ps))))
           (if ,var
               ,(c var :first-non-null
                   (car ps))
               ,(c-or0 (cdr ps)))))))

(def c-and (&rest ps)
  `(cond .,(maplist #'c-and0
                    (or ps (list t)))))

(lisp:defun c-and0 (ps)
  (if (null (cdr ps))
      `(t ,(c (car ps) :eval-all (car ps)))
      `((not ,(c-hide (car ps)))
        ,(c nil :first-null (car ps)))))

(deftype c-and (&rest b) `(and ., b))

(deftype c-or (&rest b) `(or ., b))
```

Figure 10: The code for the part of COVER that annotates conditionals.

code generated by a macro are quoted in the body of the macro.)

Figure 9 shows part of the results of annotating the function `my*` from Figure 1. The annotated definition is preceded by a call on `add-top-point`, which enters a point describing the definition into `*points*`. Within the definition, calls on `record-hit` are introduced at strategic locations. Each call contains a quoted point that is essentially a template for what should be introduced into `*points*`. The first `when` in `my*` is converted into an `if` that has cases corresponding to the success and failure of the predicate tested by the `when`. The call on `record-hit` that precedes this `if` contains a point with subpoints that establishes the cases of the `if`. This ensures that both cases of the `if` will be present in `*points*` as soon as the `if` is exercised, even if only one of the cases is exercised.

The hidden point associated with the predicate tested by the `when` establishes an appropriate context for points within the predicate itself. It is unnecessary in this example, because there are no such points. In the `cond` in the function `g` in Figure 3, a similar hidden point associated with the first predicate tested serves to correctly position the points associated with the `and` (see Figure 4).

For the most part, the macros in Figure 10 operate in straightforward ways to generate annotated conditionals. However, `def`, `c`, `c1`, and `c0` interact in a somewhat subtle way using `macrolet` to communicate the name of a superior point to its subordinates. This could have been done more simply with `compiler-let`; however, `compiler-let` is slated to be removed from Common Lisp.

Underlying approach. The annotation scheme used by COVER is designed to meet two goals. First, it must introduce as little overhead as possible when the annotated function runs. (It does not matter if the process of inserting annotation is expensive and it does not matter if the process of printing reports is expensive. It does not even matter if processing is relatively expensive the first time a point is exercised. However, it is essential that processing be very fast when an exercised point is exercised a second time.)

Second, the scheme must work reliably with interpreted code, with compiled code loaded from files, and with code that is incrementally compiled on the fly. This introduces a number of strong constraints. In particular, you cannot depend on using some descriptive data structure built up during compilation, because you cannot assume that compilation will occur. On the other hand, if you use quoted data structures as in Figure 9, you cannot make any assumptions about what sharing will exist or whether they will be copied, because some Lisp compilers feel free to make major changes in quoted lists.

To achieve high efficiency, `record-hit` (see Figure 5) alters its argument by side-effect to mark it exercised. Side-effecting a compiled constant is inherently dangerous, but is relatively safe here, because the changed value is an integer, and the point data structure cannot be shared with any other point data structure, because no two points can have the same `name`.

The first time a given call on `record-hit` is encountered, it enters the point which is its argument into `*points*`. This is done by first looking to see if the point is already there (e.g., because it was entered by an `add-top-point` or is a subordinate point that was explicitly entered as part of its superior point). If it is not there, it is copied and inserted as a subordinate point of the appropriate superior point. (By this process, `*points*` is dynamically built up in exactly the same way when executing interpreted and compiled code.) If the superior point cannot be found, nothing is done. (This can only happen when the annotation of the currently executing function has been forgotten.)

The second time a call on `record-hit` is encountered the only thing it has to do is check that the point has been exercised. If it has, nothing needs to be done. If a `cover:reset` has been done, then the check will fail, and `record-hit` relocates the point in `*points*`, and sets the `hit` flag. (This second lookup could be avoided

if the quoted point had been directly inserted into `*points*` instead of copied. However, this is unsafe for two reasons. First, the sharing would mean that side-effects to `*points*` would translate into side-effects to compiled list constants. This will cause many Lisp systems to blow up in unexpected ways. Second, in some Lisp systems compiling an interpreted function can cause the quoted lists in it to be copied. As a result, you cannot depend that any sharing set up between a global data structure and quoted constants will be preserved.)

The operation of COVER requires that each point be given a unique identifying name. The naming scheme used assumes that a given conditional form will not have two predicates that are `equal` and that a chunk of straightline code will not contain two conditional forms that are `equal`. If this assumption is violated, COVER will merge the two resulting points into one.

The power of Lisp. COVER is a good example of the power of Lisp as a tool for building programming environments. Because Lisp contains a simple representation for Lisp programs, it is easy to write systems that convert programs into other programs. Because Lisp encompasses both the language definition and the run-time environment, it is easy to write systems that both manipulate the language and extend the run-time environment. Systems like COVER are regularly written for C and other Algol-like languages; however, this is much harder to do than in Lisp.

Acknowledgments

The concept of code coverage is an old one, which is used by many (if not most) large programming organizations. COVER is the result of several years of practical use and evolution.

This paper describes research done at the MIT AI Laboratory. Support was provided by DARPA, NSF, IBM, NYNEX, Siemens, Sperry, and MCC. The views and conclusions presented here are those of the author and should not be interpreted as representing the policies, expressed or implied, of these organizations.

Obtaining COVER

COVER is written in portable Common Lisp and has been tested in several different Common Lisp implementations. The full source for COVER is shown in Figures 5, 7, 8, and 10. In addition, the source can be obtained over the INTERNET by using FTP. Connection should be made to `FTP.AI.MIT.EDU` (INTERNET number 128.52.32.6). Login as "anonymous" and copy the files shown below.

In the directory `/pub/lptrs/`

`cover.lisp`	source code
`cover-test.lisp`	test suite
`cover-doc.txt`	brief documentation

References

[1] R.C. Waters, "Supporting the Regression Testing of Lisp Programs," ACM Lisp Pointers, 4(2):47–53, June 1991.