MASSACHUSETTS INSTITUTE OF TECHNOLOGY

ARTIFICIAL INTELLIGENCE LABORATORY

A.I. Memo No. 651 November, 1981

A Program Testing Assistant

David Chapman

Abstract

This paper describes the design and implementation of a *program testing assistant* which aids a programmer in the definition, execution, and modification of test cases during incremental program development. The testing assistant helps in the interactive definition of test cases and executes them automatically when appropriate. It modifies test cases to preserve their usefulness when the program they test undergoes certain types of design changes. The testing assistant acts as a fully integrated part of the programming environment and cooperates with existing programming tools, including a display editor, compiler, interpreter, and debugger.

Keywords: debugging; program testing assistant; Programmer's Apprentice; programming environments; testing.

This report describes research done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for the laboratory's artificial intelligence research has been provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-80-C-0505, and in part by National Science Foundation grant MCS-7912179.

Table of Contents

1. Introduction 1

2. An Overview of the Testing Assistant 2
2.1 The Structure and Function of Test Cases 2
2.1.1 Test Forms and Test Data 2
2.1.2 Restoring Global State 2
2.1.3 Success and Failure 3
2.1.4 Which Test Cases Are Run 3
2.2 Bug Records 4
2.3 Chains 4
2.4 The Test Case Transformer 4
2.5 Correspondences and the Library 4

3. A Scenario 5

4. The Implementation 12
4.1 Some Success Criteria 12
4.2 The Representation of Test Data 12
4.3 The Copier 13
4.4 Expense 13
4.5 Analysis of the Programmer's Code 14
4.6 The Test Form Transformer 15
4.6.1 Syntactic Transformations 16
4.6.2 Semantic Transformations 17
4.6.3 The Inheritance of Datatype Transformations 18
4.7 Transformation of Data Objects 19

5. Related Work 20

1. Introduction

A programmer often tests his program as he writes it. As each part nears completion, he tries it on a few examples to convince himself that it is mostly correct. These test cases are typically lost, and little thought has gone into systems which aid a programmer in their full utilization. This paper describes a *program testing assistant* designed and implemented by the author which

o makes it easy for the programmer to define test cases;

o automatically runs test cases when appropriate;

o automatically modifies test cases to preserve their usefulness when the program they test undergoes certain types of design changes;

o generates test cases at the programmer's command from a library of common forms.

The testing assistant is designed to be as unobtrusive as possible. It does not require the programmer to make major modifications in the way he works. To this end, it is integrated with existing interactive programming tools, including a display editor, compiler, interpreter, and debugger. Although many of its techniques are heuristic, it operates automatically most of the time and recovers gracefully (generally by asking the programmer questions) when its heuristics are insufficient to a task.

The testing assistant analyzes the programmer's source code with his help, developing a data-abstraction view of it. The assistant organizes functions into hierarchical modules, called *layers*, and keeps track of the *datatypes* of the dataflow into and out of each function. The assistant accepts programs and test cases written in Lisp only, but its design is substantially language-independent.

My research on the testing assistant avoided areas that have already been worked on. Most previous testing research concerns generating test data that will exhaustively exercise a nearly complete program, for instance by ensuring that all controlflow paths are tested at least once. The testing assistant has no notion of exhaustive testing and no general technique for producing test data.

The next section presents a brief overview of the testing assistant. Section 3 gives an example scenario of the assistant's use, with commentary and with additional explanation of topics introduced in the overview. Section 4 examines interesting implementation issues and techniques, and the final section relates the testing assistant to other research.

2. An Overview of the Testing Assistant

2.1 The Structure and Function of Test Cases

The structure of test cases directly reflects their function. A test case consists of

- o a *test form*, which is a procedure to execute;
- o some *test data*, to which the test form is applied;
- o a *copied environment*, which is a context in which the test case is run;
- o a *saved terminal input stream*;
- o the *correct results* of running the test case, consisting of a return value and a saved terminal output stream;
- o two *success criteria*, which determine whether the test case succeeds or fails based on its results;
- o a *feature* for which the test case tests, and which is used to determine when it is run.

2.1.1 Test Forms and Test Data

A *test form* is a procedure (a lambda expression, in Lisp) which is executed to perform a test. It may be applied to *test data*; several different test cases may consist of the same test form applied to different test data.

2.1.2 Restoring Global State

Many programs reference or modify the contents of a global environment, such as a database. Test cases for such programs may need to be run with global variables in a particular state. The assistant can make a copy of the part of the global environment a test case references when it is defined and restore this global state when the test case is run later. Restoration of state protects the test case against changes in the environment that may have been made by the programmer or by its own previous runs. A generally useful copied environment can be shared among several test cases and referred to by name.

The assistant saves terminal input typed by the programmer to a test case as it is defined. This input stream is replayed when the test is run automatically.

2.1.3 Success and Failure

When the assistant runs a test case, it can either *succeed* or *fail*, depending on its results; that is, on its output to the terminal and on its return value. Failure is typically used to indicate that there is a bug in the program, although test cases could detect other conditions. Successful tests are invisible to the programmer; the assistant alerts him to tests that fail.

When the programmer defines a test case, it is run and its results are displayed. These are taken as correct unless he indicates otherwise. The results of a later, automatic run and the correct results are compared by two *success criteria*, one for the return value and one for the terminal output, which signal success or failure. The programmer can explicitly set the success criteria for a test case or use the defaults, which check for simple equality of a result and its correct value. He can also define his own success criteria, for example to check for equality of returned objects of a user-defined datatype.

2.1.4 Which Test Cases Are Run

When the programmer finishes a series of modifications of his program that leave it in a consistent state, he can ask the assistant to test it. Many of the test cases the assistant has stored may not be affected by the changes the programmer has made; it would be wasteful to run them, because they will succeed or fail just as they did before. The *feature* mechanism associates test cases with the portions of the program they test so that only those relevant to modified portions have to be run. Lisp programs are naturally divided into functions; each feature associates the test cases that test for it with a set of *suspect functions*. When the programmer asks the assistant to test his program, it runs those test cases that test for features whose suspect functions are among those modified. The programmer can also explicitly ask the assistant to test for a given feature.

An example feature which will appear in the scenario is the *multiset fault* of sets. This is a characteristic form of bug in which an object, inserted twice, remains in the set after being deleted once. Multiset behavior might be due to a bug in the insert routine (which was supposed to check that the object was not already in the set before inserting it), or to one in the delete routine (which was supposed to delete all copies of the object if there were several present); the insert and delete functions are the suspect functions for the multiset fault.

Not all test cases test for a specific bug. A test case may just exercise some portion of the program to detect problems that might be introduced by later changes. In this case, the programmer may specify a layer as a feature. All the functions in the layer are taken as its suspects. If more functions are added to the layer, they become suspect functions automatically.

2.2 Bug Records

The failure of a test usually indicates that some of the suspect functions of its feature are buggy and need to be fixed. A record can be created of the failure, which will inhibit further testing of the feature until the programmer declares that he has fixed the bug. Bug records may contain a description of the problem, which the assistant reminds the programmer of periodically. The programmer can also directly create a bug record when he notices a problem other than a failing test case, and can ask for a list of all outstanding bug records. These operations form an automated agenda for debugging.

2.3 Chains

Each time a test case is run, a new copy of its stored environment must be created to avoid making permanent changes in it. It is possible to combine several test cases together in a *chain* in which each is run in the environment left behind by the previous one in the chain. Chaining saves expensive copying operations, first because only one copy of the environment needs to be made when the whole chain is run, and second because it is often possible to avoid copying altogether by setting up an environment with an *initializing form* at the beginning of a chain.

2.4 The Test Case Transformer

Some program modifications reflect changes in the specifications of functions. These modifications may invalidate test cases, because, for example, they then call the changed functions with the wrong arguments. The assistant's *test case transformer* tries to recode test cases when program specifications change. This is extremely difficult in general, but the assistant succeeds in some common cases which account for many program changes. The assistant also recognizes the cases in which it can not help and asks the programmer to fix the affected cases manually.

2.5 Correspondences and the Library

Layer correspondences are a means of generating test cases automatically. The assistant has a *library* of useful "cliche" test cases which can be modified slightly to test the user's program. This library comes in layers organized around abstract datatypes, such as sets, vectors, and associative databases, and the functions that operate on them.

The correspondence mechanism is designed principally to exploit the library. A correspondence is much like an analogy, in the sense of Winston [6]; it provides a mapping from parts of a *source* object to a *target* object that allows knowledge about the one to be applied to the other. A layer consists of a set of functions, a set of datatypes, and a set of features; a layer correspondence maps the functions, datatypes, and features of its source layer to those of its target layer. The programmer can create a correspondence between a library layer and one of his own layers or between two of his own layers. Once a correspondence is created, test cases from the source layer are modified automatically to make them usable in the target layer.

3. A Scenario

Reading a scenario illustrating the use of the testing assistant is the best way to understand its functionality. I will take an air traffic control simulation as the program the assistant is helping develop. This program is very simple, and was written specifically to illustrate the use of the testing assistant. The testing assistant has been used to help test several larger programs, including itself.

Parts of the scenario are concerned with a set of geometric utilities which manipulate *points* and *vectors* implemented as lists, so that a vector from (100, 200) to (300, 400) is represented as (100 200 300 400). Points and vectors are created with the constructor functions make-vector and make-point respectively, both of which simply return a list of their arguments. The advantages of using these abstract datatype constructors in preference to the standard list-constructor function list will be discussed in section 4.2.

The air traffic program moves a set of *planes* through two-dimensional Cartesian space at varying velocities and keeps them from colliding. Each plane has a *course*, which is a vector. The first point in the course vector is the current position of the plane; the plane is heading toward the other point in the vector at a velocity proportional to its length. A record object of datatype plane is created with the constructor function make-plane which takes a course as its argument; the course of a plane object can be accessed with the plane-course function.

First the programmer defines the datatypes plane, course, vector, and point. He writes the function move, which updates a plane's course according to its velocity and heading. It calls the functions vector-add, vector-length, and vector-scale; the programmer makes all these the suspect functions of the feature move.

Next the programmer does some testing. In a typical Lisp system without the assistant, he would type test cases into a *read-eval-print loop*, a program which reads a Lisp expression from the keyboard, calls the interpreter to evaluate it, and prints the result. The assistant provides an augmented read-eval-print loop for defining test cases. This environment does all the same things as a standard read-eval-print loop and in addition allows the programmer to declare that an evaluated expression is a test case.

The programmer's input is in bold face and capitalized.

```
(MOVE (MAKE-PLANE (MAKE-VECTOR 0 0 100 100)))
(33 33 133 133)
```

Note that move returns the new course of the plane it moves.

```
↑Test case.
```

By typing control-T (↑T) the programmer declares that the last form evaluated is a test case. The system echoes "est case." and prompts for the feature for which this is a test.

```
Feature: MOVE
```

The assistant stores the test case along with its result for future use.

To detect collisions, the system needs to find intersections of course vectors. The programmer writes **intersection**, which depends on several auxiliary functions, such as **slope**, **intercept**, and **between?**. These are the suspect functions of the feature **intersection**.

In the next interaction, the programmer explicitly sets the feature for a series of test cases with control-F. Note that it is always possible for either the assistant or the programmer to take the initiative in communicating any piece of information, depending on whether the programmer prefers to be asked questions or to issue commands.

```
↑Feature: INTERSECTION
```

The *parameterized test case* facility makes it easy to produce many test cases with the same test form. This is useful in extensively testing a single feature. When the programmer types a lambda expression at the augmented read-eval-print loop, the assistant creates a series of parameterizations of the expression, prompting for test data to apply it to.

```
(LAMBDA (VECTOR1 VECTOR2) (INTERSECTION VECTOR1 VECTOR2))
vector1: (MAKE-VECTOR 0 0 100 100)
vector2: (MAKE-VECTOR 0 100 100 0)
(50 50)

vector1: (MAKE-VECTOR 0 0 100 100)
vector2: (MAKE-VECTOR 300 300 400 400)
nil
```

intersection returns **nil**, the special object Lisp uses to represent falsehood or the empty set, if the vectors it is passed do not intersect.

```
vector1: (MAKE-VECTOR 0 0 100 0)
vector2: (MAKE-VECTOR 100 0 100 100)

>>>>Error: Division by zero.
```

The interpreter has detected an error in the evaluation of this parameterization. It calls the debugger so the programmer can see what went wrong. The debugger is used exactly as it would have been without the assistant.

```
quotient:
   Arg 0 (number): 100
   Rest arg (numbers): (0)
→ BACKTRACE.
quotient ← slope ← line-intersection ← intersection
→ EXIT.
```

By looking up the stack, the programmer confirms his suspicion that finding the intersection of two vectors involves finding their slopes, and that the **slope** function does not handle vertical lines properly. He exits the debugger, returning control to the assistant. The assistant notes that the the evaluation caused an error:

```
Make a bug record? YES.
Bug: -SLOPE FAILS ON VERTICALS.
```

The programmer makes a bug record to remind himself to fix slope. When he has done so, the following dialog ensues:

```
DONE EDITING.
Bug in feature ''intersection'': Slope fails on verticals.
-- Fixed? YES.
```

The programmer has modified slope, a suspect function of the intersection feature, so the assistant asks if the modification included fixing the outstanding bug associated with that feature. Given that it did, it is useful to run test cases for the feature.

```
Running test cases for this feature.
((lambda (vector1 vector2) (intersection vector1 vector2))
 (make-vector 0 0 100 0)
 (make-vector 100 0 100 100)) =>
(100 0)
New result; is it correct? YES.
```

Note that although all the parameterized test cases are re-run, only the one that needs interaction with the programmer is displayed.

Using the function intersection, the programmer defines a collision-detection layer. To test it, he arranges an environment in which two planes are about to run into each other and saves it as the standard environment to test collision detection in. The control-E command prompts for the name of an environment: if it is a known name, the corresponding environment is retrieved; otherwise the assistant copies global variables selected by the programmer and gives the newly created environment the specified name.

```
↑Environment: COLLISION-ENV
[New environment.]
Copy what? *PLANES*
```

The environment will include the contents of the plane database *planes*, which the programmer has arranged with two planes on intersecting courses.

```
(RUN 1)
```

The top-level function of the simulation, run, moves each plane in turn. It takes an argument, which is the number of "ticks" or iterations to run for, and returns the symbol done.

```
<<Warning! plane-23 is on a collision course with plane-69.>>
<<Collision! plane-23 and plane-69.>>
done
↑Test case.
Feature: COLLISION
```

With collision detection apparently working, the programmer defines a collision prevention layer to correct the courses of planes so they avoid each other. He can test collision prevention in the same environment he used for testing collision detection.

```
↑Environment: COLLISION-ENV
```

The programmer retrieves his saved environment by typing control-E as before.

```
(RUN 1)
<<Warning! plane-23 is on a collision course with plane-69.>>
<<plane-23: course corrected.>>
done
(PLANE-COURSE PLANE-23)
(704 813 455 808)
↑Success criterion: VECTOR-EQUAL
```

The programmer verifies that the course of plane-23 has in fact changed. He sets the success criterion for the last form to vector-equal, which he has defined to check equality of vectors.

```
↑2↑Test cases chained.
```

The programmer marks the last expressions evaluated as a chain by typing control-2 control-T.

A more rigorous test is to see what happens when three planes are about to collide.

```
↑Environment: COLLISION-ENV
```

The programmer uses an initializing form to add another plane to the standard collision-env environment:

```
(PUSH (MAKE-PLANE (MAKE-VECTOR 650 750 650 900)) *PLANES*)
(plane-88 plane-23 plane-69)
↑Initializing form.
(RUN 1)
<<Warning! plane-23 is on a collision course with plane-69.>>
<<plane-23: course corrected.>>
<<Collision! plane-23, plane-88.>>
done
↑Wrong result.
```

Apparently the collision corrector is not smart enough; the programmer types control-W to mark this as an incorrect result.

```
Make a bug record?  YES.
Bug: FAILS ON MULTIPLE COLLISIONS.
(PLANE-COURSE PLANE-23)
(704 813 455 808)
↑Success criterion: VECTOR-EQUAL
↑2↑Test cases chained with 1 initializing form.
```

Now it's time to go home.

The programmer starts the next day by asking for a list of outstanding bug records and decides to work on the multiple collision bug. The function correct-course-to-avoid will have to be changed. It used to take as arguments a course to correct and a plane to avoid; it should take a course and a list-of-planes, all of which should be avoided. When the programmer makes this change, any test case calls on correct-course-to-avoid of the form

(correct-course-to-avoid *course plane*)

are automatically transformed by the assistant into

(correct-course-to-avoid *course* (LIST *plane*))

by applying the function list, a transformation which turns an object of any datatype into a list of one object of that datatype.

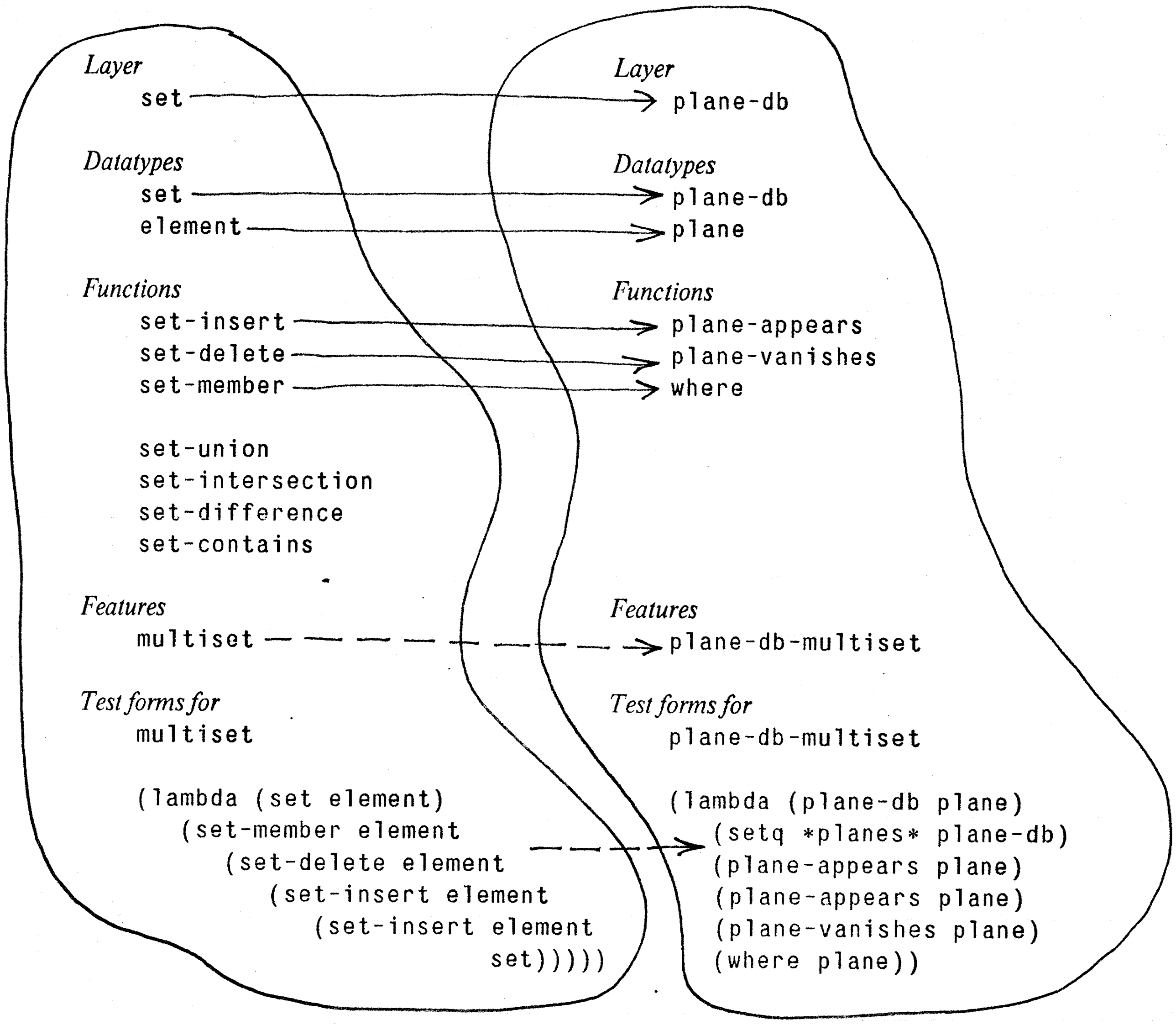
It is important to understand that this sort of transformation is applied only in test cases, and not in the programmer's code. Transformations are not correctness preserving; they are heuristic and may result in invalid or useless test cases. Nevertheless, a heuristic transformer is better than the alternative—throwing away all test cases when the functions they call change.

```
DONE EDITING.
Bug in feature ''collision'': Fails on multiple collisions.
-- Fixed? YES.
Running test cases for this feature.
(push (make-plane (make-vector 650 750 650 900)) *planes*)
(run 1)
<<Warning! plane-23 is on a collision course with plane-69.>>
<<plane-23: course corrected.>>
Output is different up to success criterion ''equal''.
Previous [incorrect] output was:
<<Warning! plane-23 is on a collision course with plane-69.>>
<<plane-23: course corrected.>>
<<Collision! plane-23, plane-88.>>
Is the new output correct? YES.
```

Next the programmer makes provision for planes entering and leaving the airspace the program is concerned with. The database *planes* contains all the "visible" planes. The function plane-appears enters a plane into the database; plane-vanishes removes one; and where returns the (x, y) position of a plane, or nil if it is not in the database.

Since this plane database behaves so much like a simple mutable set, the programmer makes a correspondence with the set layer of the library. He supplies the datatype and functional correspondences, from which the system derives new features and automatically codes new test cases.

The assistant represents the correspondence like this:



```
(lambda (set element)
   (set-member element
      (set-delete element
         (set-insert element
            (set-insert element
                        set)))))
```

```
(lambda (plane-db plane)
   (setq *planes* plane-db)
   (plane-appears plane)
   (plane-appears plane)
   (plane-vanishes plane)
   (where plane))
```

Once the correspondence has been declared, the multiset feature and its test cases are applied to the plane database layer. The assistant recodes the test form for multiset, changing the names of the functions and variables: set-insert becomes plane-appears, and so on. In fact, plane-appears has a slightly different interface from that of set-insert: set-insert is passed a set argument, whereas *planes*, which plane-appears modifies, is a global variable. These sorts of minor differences are smoothed out by the transformer. The set to plane-db correspondence is an incomplete one; set-union, for instance, is not in the correspondence. There may generally be functions and datatypes in either the domain or range of a correspondence, and test cases using these functions or datatypes can not be carried over.

The test form produced for plane-multiset in the correspondence is not a complete test case; it is a lambda expression which must be applied to a plane-db and to a plane.

```
Generated form for ''plane-db-multiset'':
(lambda (plane-db plane)
  (setq *planes* plane-db)
  (plane-appears plane)
  (plane-appears plane)
  (plane-vanishes plane)
  (where plane))
Instantiate this form? YES.
plane-db: NIL
plane: (MAKE-PLANE (MAKE-VECTOR 0 0 100 100))
Result is (0 0), OK?  NO.
```

The plane database does indeed suffer from the multiset fault. The programmer fixes the bug by modifying plane-appears, a suspect function of plane-db-multiset.

```
DONE EDITING.
Running test cases for feature ''plane-db-multiset'':
((lambda (plane-db plane)
   (setq *planes* plane-db)
   (plane-appears plane)
   (plane-appears plane)
   (plane-vanishes plane)
   (where plane))
 nil
 (make-plane (make-vector 0 0 100 100)))
=> nil
Return value is different up to success criterion ''equal''.
Previous [incorrect] value was:
(0 0)
Is the new return value correct? YES.
```

4. The Implementation

The testing assistant has been implemented on the MIT Artificial Intelligence Laboratory's Lisp Machine [1], a single-user computer optimized for Lisp. Where possible, the testing assistant makes use of existing programming tools and conventions in the Lisp Machine environment. The assistant acts as an integrated part of the Lisp Machine display editor, Zwei (a relative of the PDP-10 editor EMACS), which provides a completing reader and a uniform command syntax that are exploited by the assistant. The assistant uses the interpreter and debugger in running test cases and the compiler's expert knowledge of program syntax in its analysis of free variable references.

4.1 Some Success Criteria

The default success criterion is the Lisp function equal. Some other criteria are eq (a "stronger" form of equality); not-nil, which ignores the correct result and succeeds if the test case does not return nil; set-equal, which checks two sets to see that they contain the same elements; and isomorphic, which checks that arbitrary structures, possibly including pointer cycles, are topologically identical.

4.2 The Representation of Test Data

A test datum is stored not as a value, but as a expression which yields a value when it is evaluated. This is necessary because the value of a test datum is often relative to the environment the test case is run in. For example, the programmer might supply plane-23 as a test datum for a case whose environment includes *planes*. Each time the case is run, a new copy of plane-23 is used, so it is necessary to find the current one. A somewhat different use is to supply (random), which generates a random number, as a test datum; this will result in a different number being used each time the datum is evaluated.

Storing test data as expressions to be evaluated has the additional benefit that if the programmer uses abstract datatype constructors to specify objects, he is protected from changes in the way the datatypes are implemented. For instance, in the geometric utilities, vectors are just lists, but they are constructed with the function make-vector which makes a list of its arguments. If the implementation of vectors is changed to a record structure, the parameterized test cases created in the scenario will still work, because each test datum is stored as (make-vector *x0 y0 x1 y1*), which will now return a record object.

4.3 The Copier

Environments, unlike test data, are stored as sets of variables and copies of their actual values. This is complicated by the peculiarity of Lisp that variables are objects called *symbols*, which also can be parts of data structure. Moreover, each symbol has not only a value but also a *property list*, which sometimes stores unstructured data. Thus, an environment is in fact a set of triples, each consisting of a symbol, its value, and its property list. When a test case is run, each symbol in its environment has its value cell bound to a copy of the value specified in the corresponding triple, and its property list bound to a copy of the stored property list.

The copier works much like a garbage collector: it starts from some root node or nodes (which are global variables referenced by a test case) and traces pointers from them recursively, marking items as they are copied, until every reachable object has been marked. A difference is that typical garbage collector algorithms only leave one copy of the structure, destroying the original with forwarding pointers, whereas the assistant's copier must leave the structure being copied intact. This is done by inserting copy objects in a hash table keyed by the object they are copied from.

A database may be sufficiently interconnected that all parts of it have pointers to all other parts; in this case copying an apparently innocuous root node results in storing away the entire thing. The programmer can use a *filter* to specify that a test case references only parts of a database. A filter consists of a series of sets of objects that are either excluded from or included in the copied environment. For something to be copied, it must be in all of the included sets and not in any of the excluded ones. Things not copied are replaced by a special object that makes it easier to detect when a test case tries to reference them. The sets themselves are specified by predicates on objects; half a dozen common predicates are available from a menu supplied by the assistant, or the the programmer can give a completely arbitrary one. Forms in the menu include the value of a given symbol; the property list of a symbol, or a certain property of every symbol; a particular field of a record-structure; all the objects in a list; and all objects of a certain datatype.

In principle, terminal stream copying could be extended to input from and output to other devices (disk files, for instance). Unfortunately, the structure of the Lisp Machine input/output system makes this difficult in the general case.

4.4 Expense

Some test cases take an annoyingly long time to run, either because they do a lot of computation, or because of environment copying overhead. It is possible to label a test case *expensive*. This allows the programmer to suppress or enable the running of slow test cases at different points in the program development process. He can set one of three testing modes at any time: *Expensive*, in which expensive test cases are run just like any others; *Cheap*, in which only fast cases are run; and *Selective*, in which the system asks before running expensive test cases. The run time of a test case is measured, and if it exceeds a certain constant amount the assistant offers to mark it expensive.

4.5 Analysis of the Programmer's Code

The testing assistant is made substantially language-independent by confining language-dependent features to specific modules: analysis of the programmer's code, and analysis and recoding of test cases in the transformer. These modules could be duplicated for other languages relatively easily.

For each function in a program the assistant needs to establish its *ports* and the layer it is in. Ports are the means by which dataflow enters and leaves a function; knowledge about ports is used by the transformer. A port has a *route*, a *variable name*, and a *specification*. The route of a port encodes the way the data passes to or from the function: by a free variable, through the return value, or as an argument. The name of a port is the formal name of an argument or free variable; return values have no names. In the present implementation the specification field of a port is just the datatype of the objects that pass through it.

In determining a function's ports, the compiler is called to find references to global variables. This is the only route information not explicit in function definitions. Determining datatypes is more difficult for Lisp, because it does not have any sort of datatype declarations. Traditional datatype declarations such as integer, although perhaps heuristically useful, would not be a complete solution to the problem, because the assistant manipulates highly specific datatype information, such as "ordered list of late planes without duplicates".

Typically, the mnemonic names given variables contain significant datatype information, much of which the assistant can extract using heuristic rules. These heuristics are highly specific to Lisp and to individual programming styles and conventions, so the programmer may parameterize and modify them to suit his own idiosyncrasies. The extraction of datatypes from variable names doesn't work in cases in which a port is not named explicitly: the return value of a function being the only case in Lisp. In these cases, and in those in which the heuristics fail, the programmer can add a datatype declaration to the function; the assistant uses a simple template-driven recognizer to extract these declarations from comments.

```
;; Run the simulation for a while, returning a dont-care.
;; This function is in layer run.
(defun run (n)
  (do ((i 1 (1+ i)))
      ((= i n))
    (mapc #'move planes)
    (cond (*carefully-p* (check-collisions))))
  'done)
```

In this example, the assistant assumes that the global variable *planes* is of datatype list-of-planes, since list is the default aggregative datatype. By a Lisp convention, *carefully-p* has datatype boolean since it ends in -p. The argument n is taken to be a number because it is only one letter long. The programmer has specified that the return value is of datatype dont-care, meaning that the function is for side-effect only.

Similar techniques are applied to the names of functions to guess their return values and layers. For example, late-p returns a boolean value, and move-internal-1 is in the same layer as the function move. When a function name does not supply enough information, as is the case with run above, the programmer can use a declaration, or the assistant will ask.

4.6 The Test Form Transformer

The transformer is triggered by changes in the set of ports for a function. Transformations fall into two classes: syntactic and semantic. Syntactic transformations are ones that fix test cases after the route of a port is changed. Examples are reordering the arguments to a function and changing a function to take an input through a free variable rather than through an argument. Semantic transformations are induced by changing the datatype of a port. The semantic transformer tries to apply a *datatype transformation* to the datatype of the old port to make it match that of the new port; a datatype transformation is a function which implements a "natural" mapping from one datatype to another. The assistant has a heuristic scheme for finding the right transformation to use given the datatypes of the two ports.

The transformer is implemented as five phases: the *port correspondence matcher*, the *dataflow analyzer*, the *syntactic transformer*, the *semantic transformer* (incorporating the *transformation inheritor*), and the *coder*.

The port correspondence matcher tries to figure out what changes the programmer has made since the last time it was invoked. For each modified function, it looks at the sets of ports it had before and after the changes, and heuristically matches them on the basis of several criteria: in order of descending importance, by datatype, by name, and by route. (A port that either existed before and was deleted, or was added in the changed version, is represented specially.) Such a set of pairs of ports is called a *port correspondence*. (Each function pair in a layer correspondence is in fact also implemented as a port correspondence; this is how the transformer is able to smooth over minor differences in the behavior of corresponding functions.)

The test form transformer acts on dataflow diagrams rather than Lisp code, in order to abstract the semantics of the test form away from Lisp syntax. Besides providing language independence for the transformer, this is needed because conceptually simple transformations may involve drastic syntactic changes. A dataflow diagram is a directed acyclic graph whose vertices are functions and whose edges are labeled with ordered pairs of ports. (There is an example dataflow diagram on page 17.)

The dataflow analyzer symbolically evaluates a test form and produces as output a dataflow diagram. Since dataflow diagrams do not represent controlflow, the analyzer can only operate on test forms that are straight-line code, without control constructs such as loops or conditionals. Waters's knowledge based program editor system [5] uses dataflow diagrams augmented with controlflow information to analyze and generate code involving complex control structure. It will eventually be used in place of the comparatively simple analyzer and coder I wrote.

The coder (actually the last stage of the transformer) is the inverse of the analyzer: it takes a dataflow diagram modified by the syntactic and semantic transformers and turns it back into Lisp code. The transformer does not preserve syntactic style; it may produce quite different looking code from that analyzed, even if a null transformation is applied. This is acceptable because its output is still readable and the syntax of test forms is not important.

4.6.1 Syntactic Transformations

The syntactic transformer modifies the diagram produced by the analyzer to reflect the changes expressed by the port correspondence. It substitutes the new ports in the correspondence for the old ones. This induces changes in route when the diagram is recoded. Since route is not represented explicitly in the dataflow diagram, but is a property of ports, the coder does all the real work.

Suppose check-course-collisions is modified so that the set of planes is passed to it as an argument instead of via the global variable *planes*. This is a typical syntactic transformation, accomplished simply by substituting a new port with an argument route for the old *planes* free-variable port. The following diagram makes the port correspondence clear; the syntax used is ⟨*name*, *route*, *datatype*⟩.

Ports of check-course-collisions

```
Before                                      After
<course, <argument 1>, course> ——————> <course, <argument 1>, course>
<*planes*, free, list-of-planes> ————> <planes, <argument 2>, list-of-planes>
```

When the coder is invoked, it needs to find something that will evaluate to the value of *planes* to put into the second argument position; "*planes*" itself does, so

```
    (check-course-collisions course)
```

is recoded in test cases as

```
    (check-course-collisions course *PLANES*)
```

When the opposite modification is made, so that the argument becomes a free variable input, a call of the form

```
    (check-course-collisions course planes)
```

is transformed into

```
    (SETQ *PLANES* planes)
    (check-course-collisions course)  .
```

4.6.2 Semantic Transformations

The semantic transformer looks for ports that have changed datatype; these are the ones to which semantic transformations must be applied. It calls the transformation inheritor to find a transformation between the new and the old datatype and splices a new vertex into the graph corresponding to that transformation function.

Suppose the function move, which used to take a plane argument, is changed to take a course argument.

```
<plane, <argument 1>, plane> ———> <course, <argument 1>, course>
```

The function plane-course extracts the course field of a plane object, so it is the right transformation to use.

```
    (move plane)
```

is transformed into

```
    (move (PLANE-COURSE plane))
```

New ports must be treated somewhat differently; since they are not paired with another port, there is nothing to apply a transformation to. The programmer can either specify that a new port can be derived from some other port by applying a transformation, or can give a *default form* to use for the port. Suppose move is further modified to take a velocity argument for efficiency reasons; perhaps its caller computes the velocity for some other purpose. A velocity can be derived from a course with vector-length and supplied as the second argument to move. The dataflow diagrams for this example are

Before

$$\begin{tikzcd}
\textit{plane} \arrow[d] \\
\text{plane-course} \arrow[d] \\
\text{move} \arrow[d] \\
{}
\end{tikzcd}$$

After

$$\begin{tikzcd}
\textit{plane} \arrow[d] & \\
\text{plane-course} \arrow[dd, bend right] \arrow[dr, bend left] & \\
& \text{vector-length} \arrow[dl, bend left] \\
\text{move} \arrow[d] & \\
{} &
\end{tikzcd}$$

and the latter is coded as

```
    (LET ((COURSE-1 (plane-course plane)))
      (move COURSE-1 (VECTOR-LENGTH COURSE-1)))
```

Because there are two references to the return value of the call on plane-course, the coder is forced to introduce a local variable course-1 to store it in. The name course-1 is generated from the datatype of the corresponding dataflow edge (plane-course returns an object of datatype course) and a numeric suffix which guarantees that the name has not been used elsewhere. This non-local change, which, though semantically simple, involves major syntactic modifications, is an illustration of the utility of the dataflow representation.

4.6.3 The Inheritance of Datatype Transformations

Datatypes are arranged in an inclusion hierarchy, and transformations can be inherited through the hierarchy. For instance, the function vector-length transforms a vector into a length; since a course is a kind of vector and a length is a kind of number, vector-length also transforms a course into a number. In general, if a transformation is needed from some *source* datatype to some *target* datatype, it is safe to search for a transformation from any datatype that is a generalization of the source to a datatype that is a specialization of the target.

$$\begin{tikzcd}
& \text{[\textit{generalization of source}]}\ \texttt{vector} \arrow[dd, Rightarrow] & \\
\text{[\textit{source}]}\ \texttt{course} \arrow[ur] \arrow[rr, Rightarrow] & & \texttt{number}\ \text{[\textit{target}]} \\
& \texttt{length}\ \text{[\textit{specialization of target}]} \arrow[ur] &
\end{tikzcd}$$

(Single arrows denote inclusion and double arrows transformations. The horizontal transformation can be inherited from the vertical one.)

More complicated schemes are needed for dealing with *compound datatypes* in which an aggregative datatype (such as list, array, or set) is further constrained to have elements of another datatype. This yields new datatypes like list-of-planes or array-of-sets-of-points. Compound datatypes are represented specially internally because different inheritance rules and different types of transformations apply to them. For example, an object may be turned into a list-of-objects by applying list; and therefore a plane (which is a kind of object) can be so transformed into a list-of-planes. This is a different sort of inheritance than that explained before because a list-of-objects is not in general a list-of-planes.

$$\begin{tikzcd}
\texttt{object} \arrow[r, Rightarrow, "\textit{list}"] & \texttt{list-of-objects} \\
\texttt{plane} \arrow[u] \arrow[r, Rightarrow, "\textit{list}"] & \texttt{list-of-planes} \arrow[u]
\end{tikzcd}$$

The function list is an example of a *singleton* transformation, which creates an aggregate of one element; its opposite is the *selection* transformation first which transforms a list-of-objects into an object. Aggregative datatypes often are provided with an *iterator*, which applies a function to each of their elements; mapcar is the iterator for lists. A list-of-planes can be turned into a list-of-courses by iterating plane-course over it with mapcar.

Transformations applied automatically by the assistant are intended to be "natural" so that the meaning of the test case will usually be preserved. Unfortunately, selection transformations are not invertible, and most of the object one is applied to will get lost. Test forms produced by applying selection transformations may not be very useful, though they probably will not cause errors in evaluation.

There are instances in which there is more than one inheritable transformation between two datatypes, and cases in which there are none. In these cases, the user can specify a transformation. Test case recoding can be made subject to the approval of the programmer if he likes, and he can directly edit test cases when the transformer fails.

In order to give the transformation inheritor leverage, the programmer should use variable names that contain complete datatype specifications. Still, the system can operate with reduced functionality without this information; it does not force the programmer into a rigid data-abstraction programming methodology.

4.7 Transformation of Data Objects

Techniques very similar to those used for transforming test forms are used to transform record objects, such as those stored in environments. In Lisp Machine Lisp, the defstruct special form is used to define record datastructures. The assistant analyzes defstructs and notices when they are changed. When the programmer edits a defstruct definition, a *field correspondence* is created between the old and the new fields, just as a port correspondence is created between the new and old ports of a function. The datatypes of the fields can often be guessed from their names. If there are objects of the datatype defined by the defstruct in copied environments, the transformer modifies them by adding, reordering, or removing fields as necessary. Again, where field datatypes have changed, datatype transformations may be applied; and if there is a new field, it may be calculated from existing fields or set to a default.

5. Related Work

The testing assistant is quite different from most of the work that has been done in the software testing field in that it is less concerned with the content of the test cases themselves than with an environment to support their use. (For examples of testing research, see [7], which has a collection of papers on testing.)

One system that does focus on incremental, interactive program testing is Lieberman's TINKER [2]. TINKER is a program synthesis system using a modified programming by example strategy in which the examples are not input/output pairs but code fragments. These code fragments also serve as test cases for the program being developed. Whenever the program is edited, all the test cases are re-run. TINKER was mainly intended to explore a novel synthesis scheme; it has none of the features of the testing assistant that make testing of conventionally constructed programs easy. One useful feature of TINKER which the testing assistant lacks is a technique for automatically producing dummy subprocedures to test partially written code. This feature encourages top-down debugging, rather than the usual bottom-up style.

Some research has been done on the understanding of specific types of bugs. Shapiro's "Sniffer" system [4] has a deep understanding of a few bugs. It can recognize them from their symptoms and provide a detailed diagnosis of the cause of the problem. The testing assistant has a shallow understanding (provided by test cases in the library) of many different bugs.

Systems similar to my testing assistant are used in industry. One tester I know of is used in-house by a large firm to test microcomputer programs written in a low level language. It provides input and output stream copying and checks output for equality with that stored. It also has a facility similar to features for running only useful test cases.

The testing assistant will be integrated into the Programmer's Apprentice system described in [3]. The main limitation on the transformer's power is the restriction of port specifications to a datatype. The Programmer's Apprentice will provide a more complete specification language and powerful "cliche-based" analysis and synthesis techniques. These will greatly extend the transformer's capabilities, and will make possible more sophisticated ways of generating test cases than the current correspondence mechanism.

Acknowlegements

Support for this research was provided in part by the MIT UROP Program.

The Lisp Machine Group has implemented a programming environment of unparalleled power and beauty, which provided both an example and a base to work on.

I would like to thank Dan Brotsky, Scott Cyphers, Ken Haase, David Levitt, Neil Mayle, David McAllester, Kent Pitman, Chuck Rich, and Dick Waters, who read versions of the manuscript and provided many useful suggestions.

Dr. Charles Rich and Dr. Richard C. Waters supervised this research. They were unfailingly enthusiastic about the testing assistant, and many of the ideas in it are theirs.

References

[1] R. Greenblatt, T. Knight, J. Holloway and D. Moon, "A Lisp Machine", *Fifth Workshop on Computer Architecture for Non-numeric Processing*, Pacific Grove, CA, ACM SIGIR Notices Vol. 15, No. 2, ACM SIGMOD Record Vol. 10, No. 4, March, 1980.

[2] H. Lieberman and C. Hewitt, "A Session with TINKER: Interleaving Program Testing with Program Design", *Proc. of the 1980 Lisp Conference*, Stanford University, August 1980, August, 1980.

[3] C. Rich and R.C. Waters, "Abstraction, Inspection and Debugging in Programming", MIT Artificial Intelligence Laboratory Memo No. 634, June, 1981.

[4] D. Shapiro, "Sniffer: a System that Understands Bugs", (M.S. Thesis), MIT Artificial Intelligence Laboratory Memo 638, June, 1981.

[5] R.C. Waters, "The Programmer's Apprentice: Knowledge Based Program Editing", *IEEE Transactions on Software Engineering*, January, 1982.

[6] P. H. Winston, "Learning and Reasoning by Analogy: the Details." Formerly titled "Learning by Understanding Analogies." MIT Artificial Intelligence Laboratory Memo No. 520, April 1979.

[7] *IEEE Transactions on Software Engineering*, May, 1980: special issue on testing.