Testing Generic Ada Packages with APE

Daniel Hoffman* Jayakrishnan Nair† Paul Strooper‡

April 30, 1998

Abstract

Despite substantial research on methods and tools for testing reusable modules, little help is available for the tester in the field. Commercial tools for system testing are widely available, but tools for module testing are hard to find. This paper presents a practical approach to testing Ada packages using the Ada Package Exerciser (APE). The APE tool generates test drivers for Ada packages from test scripts written by the tester. The generated test drivers provide test inputs and check output correctness automatically, so that it is practical to rerun the tests after every change to the package implementation or its environment. The testing approach and the APE tool are described in detail, and illustrated with a simple example and a commercially developed package. Specialized techniques for testing generic packages and for performing random testing are presented.

1 Introduction

The fundamental goal of our research is to improve system quality and reduce maintenance costs through systematic module testing. While system testing is usually emphasized, module testing is also important. It is typically difficult to thoroughly test a reusable module M while it is linked into a given application. M's subprograms are often not directly accessible, some of its subprograms may not be called at all, and errors in other modules may appear to be errors in M. Conversely, errors in M may be masked by errors in other modules.

Using test drivers, a module may be tested in isolation from any particular application. Although implementing test drivers manually is straightforward, it is also time-consuming, repetitive, and error-prone, and it produces code that is costly to maintain. As a result, test driver generation is a good candidate for automated support.

In this paper, we present the Ada Package Exerciser (APE), which generates batch test drivers for Ada packages from test scripts written in a simple test language. The test scripts are developed manually by the tester, but the generated drivers run fully automatically.

*Dept. of Computer Science, Univ. of Victoria, P.O. Box 3055 MS7209, Victoria, B.C., V8W 3P6 Canada, dhoffman@csr.uvic.ca

†Dept. of Computer Science, Univ. of Victoria, P.O. Box 3055 MS7209, Victoria, B.C., V8W 3P6 Canada, jk@csr.uvic.ca

‡Dept. of Computer Science and Electrical Engineering, The Univ. of Queensland, Brisbane, 4072 Australia, pstroop@csee.uq.edu.au

APE is an Ada version of the PGMGEN testing tool [8, 10], which generates test drivers for the testing of C modules. The language support for reusable modules in Ada in the form of packages means that a tool such as APE is more widely applicable than a tool such as PGMGEN, which depends on customized "modules." For example, the mechanism for signaling and handling exceptions in PGMGEN is non-standard.

After reviewing the literature in Section 2, we introduce a generic stack package that is used as an example in Section 3. We present the APE script language in Section 4, and explain how batch drivers are generated from test scripts. In Section 5, we discuss how we typically develop an APE script by using the stack package as an example, and also explain how we test generic packages with different data types. Section 6 discusses the application of APE to a commercially developed and tested package.

2 Related work

Early work on unit testing has focused on testing software modules. For example, Panzl [15] discusses the regression testing of Fortran subroutines. The DAISTS [7], PGMGEN [8, 10], and Protest [9] systems all automate the testing of modules using test cases based on sequences of calls.

More recently, work on unit-level testing has focused on class testing in object-oriented languages [3]. Fiedler [4] describes a small case study on testing C++ objects. Frankl [5] has developed a scheme for class testing using algebraic specifications. The ACE tool [14] (like APE, ACE is an enhancement of PGMGEN) supports the testing of Eiffel and C++ classes and has seen substantial industrial use. In the ClassBench methodology [11, 12], a class is tested using a testgraph, which partially models the states and transitions of the class under test state/transition graph, and a customized test oracle, which verifies the behavior of the class under test.

In previous work on testing Ada programs, Gallagher and Narasimhan [6] describe a system for generating test data for Ada programs based on numerical optimization techniques. Barbey and Buchs [2] discuss the generation of a test set for an Ada abstract data types (ADT) from a formal specification of that ADT. Their method was devised for the testing of ADTs implemented in Ada 83, and Barbey [1] discusses how the object-oriented features of Ada 95 impact on this method. Similarly, Waterman [16] discusses the impact of Ada 95 on the testability of safety-related systems, with an emphasis on unit testing.

A number of commercial tools exist to support the testing of Ada programs. The one that most closely resembles APE is AdaTEST from IPL [13]. This tool is more general than APE in that it supports test execution, as well as coverage analysis and static analysis. It is more limited, however, in its support for test driver generation. It does provide a test harness for test execution, and a language for expressing test cases. However, the script language for APE is much simpler; APE derives its power by allowing the tester to include arbitrary Ada fragments inside APE scripts. Thus, APE leverages the tester's knowledge of Ada.

```
generic
   maxsize: integer;
   type item is private;
package stack is
   empty, full : exception;

   procedure init;
   procedure push (i : in item);
   procedure pop;
   function top return item;
   function depth return integer;
private
   stack : array (1 .. maxsize) of item;
   siz : integer;
end stack;
```

Figure 1: *stack* package specification

3 A simple example

In this paper, we assume that a module is implemented as an Ada package, which can be accessed only through its subprograms. The package specification declares the names of the subprograms, their parameters and return value types, and the names of the exceptions that the package may generate. Any constants and types provided by the package are also declared. We illustrate these ideas on a simple *stack* package, shown in Figure 1. *stack* is a generic package with two parameters: `maxsize` specifies the maximum size of the stack, and `item` specifies the type of the items stored in the stack. The subprograms of *stack* are:

- `init` initializes the stack to the empty stack.
- `push(i)` pushes i onto the stack. Exception `full` is raised if there are already `maxsize` elements in the stack.
- `pop` removes the top element from the stack. Exception `empty` is raised if the stack is empty.
- `top` returns the top element of the stack without deleting it. Exception `empty` is raised if the stack is empty.
- `depth` returns the number of elements in the stack.

4 Test program generation with APE

4.1 Test script language

The APE test script language is based on *traces*, where a trace is a sequence of subprogram calls. For example, consider the following traces for *stack* (when writing traces, we separate adjacent calls with a period).

```
init.push(10)
init.pop
```

The first trace initializes the package and pushes the value 10 onto the stack. In the second trace, the `pop` call should generate the `empty` exception because `init` reinitializes the stack to empty.

Our test cases are described by providing a trace and associating it with some aspect of the required behavior of the package in response to that trace. We represent a test case as a five-tuple

$$\langle \; trace, \; expexc, \; actval, \; expval, \; type \; \rangle$$

with the following meanings:

trace: a sequence of subprogram calls used to exercise a package.

expexc: the name of the exception that *trace* is expected to generate or `noexc` if no exception is expected.

actval: an expression, typically a function call, to be evaluated after the trace, and whose value is taken to be the actual value of the trace.

expval: the value that *actval* is expected to have.

type: the data type of *actval* and *expval*.

Below are two test cases, based on the traces described above. In test cases developed solely for exception checking, the *actval*, *expval*, and *type* fields contain `dc` (don't care).

```
<init.push(10), noexc, top, 10, integer>
<init.pop, empty, dc, dc, dc>
```

The first test case pushes 10 onto the stack and then checks that `top` returns the correct value. The second test case checks that `pop` correctly signals the `empty` exception for an empty stack.

To provide a test language powerful enough to describe the test cases, but which is easy to learn, Ada code may be freely embedded in test scripts. Code delimited by `{%` and `%}` can be placed in a variety of places in the test script, e.g., in the trace, actual value, or expected value position of a test case, or between test cases. Thus, there is no need in the script language for functions and iteration constructs—these are available in Ada and are presumably understood by the test programmer.

For example, the following test script fragment executes the test case inside the loop 100 times.

```
{% for i in 1..100 loop %}
   <init.push(i), noexc, top, i, integer>
{% end loop; %}
```

```
PackagePrefix
   "st"
ExceptionPrefix
   "st"
Subprograms
   <init,push,pop,top,depth>
Exceptions
   <empty,full>
GlobalCode
{% %}
LocalCode
{% package st is new stack(100,integer); %}
Cases
<init.push(10), noexc, top, 10, integer>
<init.pop, empty, dc, dc, dc>
```

Figure 2: Simple *stack* test script

In each iteration, the value of the loop index is pushed onto the stack, and then the element returned by `top` is checked. Clearly this test script fragment is not that useful in practice and was only used to illustrate the use of embedded code. As such, it will not appear in the full *stack* test script in Section 5.

Figure 2 shows an APE test script for *stack* containing the two test cases discussed above. The `PackagePrefix` section defines the package prefix, which APE places in front of every subprogram. The `ExceptionPrefix` section defines the prefix that APE places in front of every exception. The `SubPrograms` and `Exceptions` sections define the list of subprograms and the exceptions of the package. The `GlobalCode` section contains global Ada code (omitted in this example). APE places this code at the top of the generated test driver. The `LocalCode` section contains code that is placed in the driver after the specification of the driver procedure. The test programmer can use the `GlobalCode` for `with/use` clauses. The `LocalCode` can be used for stubs, local variables, and utility functions that are used in the test cases. In this case, the `LocalCode` defines the variable `st` that holds an instance of the *stack*. Finally, the `Cases` section contains the test cases proper.

4.2 Test Program Generation

Although implementing test drivers manually is straightforward, it is also time-consuming, repetitive, error-prone, and produces code that is costly to maintain. As a result, test driver generation is a good candidate for automated support. In this section we briefly describe how APE accomplishes test driver generation.

The APE system flowchart is shown in Figure 3—ovals indicate human readable files and boxes indicate executable programs. The test script for package P is prepared in a file

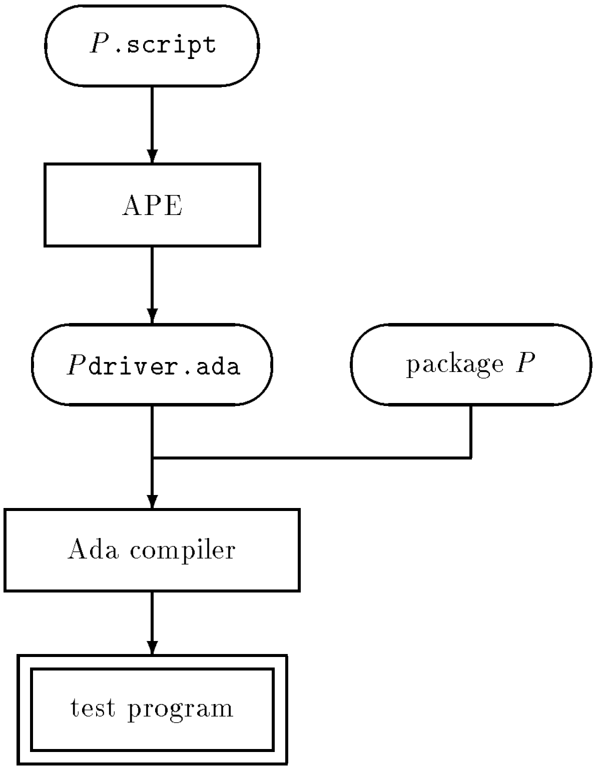

Figure 3: APE system flowchart

P`.script` by the test programmer. APE reads that script and generates an Ada driver in the file P`driver.ada`. This driver is then compiled and linked with the implementation of P. When the resulting test program is executed, the test cases in the file P`.script` are run against the implementation and any errors are reported.

To generate the test driver, APE first generates code to record exception occurrences. Then, for each test case of the form:

$$\langle\ c_1.c_2.\cdots.c_N,\ \mathit{expexc},\ \mathit{actval},\ \mathit{expval},\ \mathit{type}\ \rangle$$

code is generated to:

invoke $c_1.c_2.\cdots.c_N$
compare the actual occurrences of exceptions to *expexc*
if there are any differences
    print a message
else
    if $\mathit{actval} \neq \mathit{expval}$
        print a message
    if exceptions have occurred since c_N was invoked
        print a message

Assumptions
    `maxsize` > 2
Special values
    Module state
        size of stack: $\{0, 2, \texttt{maxsize}\}$
Access routine parameters
    none
Test cases
    for each $i \in \{0, 2, \texttt{maxsize}\}$
        load stack with $10, 20, \ldots, 10 \times i$
        call `push`, `pop`, `top`, `depth`
        check return values of `top` and `depth`
        check exception behavior of `push`, `pop`, and `top`

Figure 4: *stack* test plan

update summary statistics

Each test case generates a number of Ada statements (typically 15–20) enclosed in a begin/end pair. As it appears, embedded code is entered into the generated code. Thus, when a for loop appears above a test case, all of the code generated by the case will be in the loop body. Following the last case, code is generated to print summary statistics.

5 APE Methodology

Before writing a test script, it pays to develop a *test plan*, which describes the strategy for selecting and executing the tests, and any assumptions on which the testing depends. The test plan allows review of the adequacy and feasibility of the testing before the script is written and supports maintenance afterwards.

Test selection is done manually, based on the principles of functional testing. For a module P, critical values for P's internal state are determined. For each subprogram p, and parameter x, critical values for x are chosen. Ideally, we would test all calls with all combinations of critical parameters and internal state values. When this approach results in too many test cases, we reduce the number of test cases based on intuition and experience. One of our goals, however, has been to increase the number of test cases that can be tested economically.

The test plan for the `stack` is shown in Figure 4. For `stack`, internal values are easily controlled by procedures and observed with function calls. Test data selection for `stack` focuses on the number of elements pushed onto the stack. Special values are 0 and `maxsize`, because they represent an empty and a full stack, and one value in between, for which we have chosen 2 in this case. For each of these values, we consider normal and exceptional behavior.

Figure 5 contains a test script implementing the test plan in Figure 4. The `PackagePrefix`,

```
LocalCode
{%
        package st is new stack(100,integer);
        i: integer;

        procedure load (n: in integer) is begin
                st.init;
                for i in 1..n loop st.push(i*10); end loop;
        end load;
%}
Cases
<load(0).push(10), noexc, top, 10, integer>
<load(0).push(10), noexc, depth, 1, integer>
<load(0).pop, empty, dc, dc, dc>
<load(0).{%i:=st.top;%}, empty, dc, dc, dc>
<load(0), noexc, depth, 0, integer>

<load(2).push(30), noexc, top, 30, integer>
<load(2).push(30), noexc, depth, 3, integer>
<load(2).pop, noexc, top, 10, integer>
<load(2).pop, noexc, depth, 1, integer>
<load(2), noexc, top, 20, integer>
<load(2), noexc, depth, 2, integer>

<load(100).push(0), full, dc, dc, dc>
<load(100).pop, noexc, top, (100-1)*10, integer>
<load(100).pop, noexc, depth, 100-1, integer>
<load(100), noexc, top, 100*10, integer>
<load(100), noexc, depth, 100, integer>
```

Figure 5: *stack* test script

```
LocalCode
{%
        package st is new stack(100,Complex.Number);
        i: Complex.Number;

        function e2i (Val : in Complex.Number) return Integer is begin
            return (Complex.Real_Part(Val)*1000+Complex.Imaginary_Part(Val));
        end e2i;

        function i2e (Val : in Integer) return Complex.Number is
           R, I : Integer;
        begin
           I := Val mod 1000; R := (Val - I) / 1000;
           return Complex.Make (R, I);
        end i2e;

        procedure load (n: in integer) is begin
                st.init;
                for i in 1..n loop st.push(i2e(i*10)); end loop;
        end load;
%}
Cases
<load(0).push(i2e(10)), noexc, e2i(top), 10, integer>
<load(0).push(i2e(10)), noexc, depth, 1, integer>
<load(0).pop, empty, dc, dc, dc>
<load(0).{%i:=st.top;%}, empty, dc, dc, dc>
<load(0), noexc, depth, 0, integer>

<load(2).push(i2e(30)), noexc, e2i(top), 30, integer>
<load(2).push(30), noexc, depth, 3, integer>
<load(2).pop, noexc, e2i(top), 10, integer>
<load(2).pop, noexc, depth, 1, integer>
<load(2), noexc, e2i(top), 20, integer>
<load(2), noexc, depth, 2, integer>

<load(100).push(i2e(0)), full, dc, dc, dc>
<load(100).pop, noexc, e2i(top), (100-1)*10, integer>
<load(100).pop, noexc, depth, 100-1, integer>
<load(100), noexc, e2i(top), 100*10, integer>
<load(100), noexc, depth, 100, integer>
```

Figure 6: *stack* script for complex numbers

`ExceptionPrefix`, `Subprograms`, `Exceptions`, and `GlobalCode` sections are the same as in Figure 2. `load` is a utility function declared in the `LocalCode` of the test script; it takes an integer n as a parameter and pushes the elements $10, 20, \ldots, 10 \times n$ onto the stack. For each special value for the number of elements pushed onto the stack, there are a number of test cases that exercise a stack of that size. There are five test cases for an empty and full stack, and six for a partially full stack. When the code generated by the stack script in Figure 5 is run, it executes the 16 test cases giving 100% statement coverage of the stack implementation.

5.1 Testing generic packages

Ada generics allow a programmer to implement a package independently of the type it manipulates. The *stack* described in Section 3 is implemented as a generic Ada package that takes the maximum size and the element type as a parameter. A naive approach to testing generic packages would require a different test script for each element type. Development costs would be low because each new test script could be quickly adapted from the integer script. Maintenance costs would be high, however, because each change would have to be applied manually to each script.

We can use a single test script for all element types if we represent each element type as an integer. With the introduction of the two functions `i2e` (integer-to-element) and `e2i` (element-to-integer) this can be done with minimal effort. To test a generic stack package that stores elements of type e, the function `i2e` will take an integer parameter and return an element of type e. Similarly, the function `e2i` will take an element of type e and return an element of type integer.

For example, suppose that we are testing a stack of complex numbers, implemented as an Ada package. We define mappings from the complex number $a + b\mathtt{i}$ to the integer $n = 1000a + b$, and vice versa. Specifically, the function `i2e` takes an integer as a parameter and returns the corresponding complex number, and the function `e2i` takes a complex number as a parameter and returns the corresponding integer. Suitable restrictions on the values of a and b make these mappings one-to-one, and still permit thorough testing. The test script for stack using this technique is shown in Figure 6. With this technique the effort to modify a script to test a generic package with a different element type typically requires less than 10 lines of code.

6 Case study : `Abst_Simple_Map_Generic`

6.1 Interface overview

The `Abst_Simple_Map_Generic` (hereafter *map*) package implements a function on objects of one type, the domain, yielding objects of a second type, the range. A map thus defines a dynamic collection of bindings from the domain to the range; an arbitrary number of bindings can be created, modified, and destroyed over the lifetime of a map. The domain and the range are typically different types, although they may be the same type. For every object of the domain, there can exist no more than one object of the range. The converse is not true; a range object can be associated with one or more domain objects

```
The_Name : in String;
type Domain is private;
type Range_Of_Map is private;
Default_Range : in Range_Of_Map;

The_Number_Of_Buckets : in Baty_System_Types.Initial_Maximum;
with function Hash(The_Domain: in Domain)
                  return Baty_System_Types.Hash_Value;

The_Subsystem_Id: Baty_Subsystem.Id;
Error_Severity_When_Domain_Not_Bound: in
  Baty_Event_Handler_Interface.Severity :=
  Baty_Event_Handler_Interface.Serious_But_Not_Fatal;
```

Figure 7: *map* generic parameters

Map is an generic Ada package which takes the parameters shown in Figure 7. `The_Name` is the name of the map, which is used in error reporting. `Default_Range` is returned by the function `View_Of` when the domain specified as the parameter is not found in the map. Because *map* is implemented with a hash table, a hash function (`Hash`) and a bucket count (`The_Number_Of_Buckets`) must be provided. For a given domain object, `Hash` must always yield the same key. `The_Subsystem_Id` is the name used by the private memory manager in the *map* and `Error_Severity_When_Domain_Not_Bound` is the default severity with which errors are reported.

Map provides the operations shown in Figure 8. `Bind(`d,r,m`)` adds the pair $\langle d, r\rangle$ to the map m. If $d \in m$, then `Serious_But_Not_Fatal_Error` is raised. If *map* runs out of storage space, then `Wild_And_Fatal_Error` is raised. `Unbind(`d,m`)` deletes the entry with domain value d from m. If there is no such entry in m, then `Serious_But_Not_Fatal Error` is raised. `Bound(`d, m`)` returns true if $d \in m$ and false otherwise. `View_Of(`d,m`)` returns the range value associated with d. If there is no such entry then an error with the instantiated severity is raised, the default being `Serious_But_Not_Fatal_Error`. `Iterate_Generic` and `Iterate_With_State_Generic` each take a procedure p as a generic parameter. p is applied to each entry visited in *map*, without modifying *map*. Procedure `Process` in `Iterate_With_State_Generic` takes an extra parameter, s.

6.2 Automated testing

Before developing our own test driver for *map*, we studied a driver written by an experienced industrial tester. This driver was carefully written and documented, and achieved 100% statement coverage of the code. However, it also had some drawbacks, commonly found in even the best industrial test drivers we have seen:

```
procedure Bind(d : in Domain; r : in Range_Of_Map; m : in out Object_Ref);
procedure Unbind(d : in Domain; m : in out Object_Ref);
procedure Bound(d : in Domain; m : in Object_Ref);
procedure View_of(d : in Domain; m : in Object_Ref);
procedure Iterate_Generic(m : in Object_Ref; p : in Process);
procedure Iterate_With_State_Generic(m : in Object_Ref; p : in Process;
     s : in out State);
```

Figure 8: *map* subprograms

.

for $n \in \{0, 1, 2, 5, 500\}$
    Exception testing:
    `load(`n`)`
    for i in $1, \ldots, n$
        Check that `Bind(`i`,`$2 \times i$`)` raises an exception
        Check that `Unbind(`$n+1$`)` raises an exception
        Check that `View_Of(`$n+1$`)` raises an exception

    Normal case testing:
    `load(`n`)`
    Invoke `Iterator_Generic`; Check the sum of the range values.
    Invoke `Iterator_With_State_Generic`; Check the sum of the range values.
    for i in $1, \ldots, 500$
        if $i \leq n$
            Check that $i \in map$
            Check that `View_Of(`i`)` gives the correct range value
            Check that `Unbind(`i`)` removes the element with domain value i
        else
            Check that `Bound(`i`)` returns false

Helper Routine
`load(`n`)`
    `Destroy` the map
    for $i \in 1, \ldots, n$
        `Bind(`i`,`$2 \times i$`)`

Figure 9: Partial test plan for programmatic *map* test script

- it was lengthy and expensive to develop,
- it was difficult to enforce standardization between it and other drivers, resulting in high maintenance costs, and
- it missed many interesting combinations of special values.

These drawbacks are difficult to avoid without considerable upfront investment in methodology and tool development. In practice, testers can rarely afford to make this investment.

With an APE script, it is possible to test *map* much more thoroughly with less effort. Using loops in embedded code, a compact script can generate tests covering all the combinations of special values. A partial test plan for such a *map* test script is shown in Figure 9. Test case selection focuses on the number of elements in the map. For each of these values we consider both the normal and the exceptional behavior. The testing of exceptions is done by trying to `Bind` domain values that are already bound, and trying to `Unbind` and get the `View_Of` values that are not currently bound.

The normal case tests begin by exercising the iterators. To simplify output checking, the `Process` function passed as a parameter merely computes the sum of the domain values. If the sum is correct, it is very likely that the iterator implementation has visited every *map* element exactly once. We frequently use such spot checks to produce "pretty good output checking" at very low cost. Frequently, the only practical alternative is to omit the test entirely. The remaining normal case tests are inside a loop, with i ranging from 1 to 500. When $i \leq n$, i should be bound. The script checks that this is so, and that `View_Of` returns the correct range value. Then the script unbinds i and checks that the unbind was successful. When $i > n$, i should not be bound; the script checks that this is so.

The full test script based on this test plan is given in the Appendix. The script uses `Complex Number` as the domain type and `Integer` as the range, and uses the `e2i/i2e` technique described in Section 5.

6.3 Results

The results of applying APE methodology to the *map* package show that an APE script can achieve the same level of coverage as a handcoded driver with a substantial reduction in the lines of code that need to be written (see Table 1). Also the number of tests that are run using APE is much larger than that of the handcoded driver.

Table 2 shows the statement coverage obtained against the number of elements in the map. When we say the map has n elements, it means the map contains elements from 1 to n. Note that even with 0 elements we already get 69% coverage. However, even though we get close to 100% coverage with a small number of elements, we need a larger number to actually reach 100% coverage.

7 Conclusions

A practical and widely applicable approach for module testing has been presented. APE is most effective for packages accessed through subprogram calls. When package testing

	number of tests	LOC	statement coverage (%)
industrial driver	36	579	100
APE script	7556	123	100

Table 1: Comparison of hand coded driver driver and APE script

number of elements	statement coverage (%)
0	69
1	97
2	97
5	97
500	100

Table 2: Number of elements and statement coverage

is done as described above, information hiding tends to reduce test case maintenance. Because information hiding reduces changes to package interfaces, changes to tests based on those interfaces are also reduced. Cases selected on the basis of package implementations will still be sensitive to change. However, most cases are based on the interface alone—these will change only when the interface does.

Preliminary results, following experimental application of APE to some commercially developed packages, indicate that APE test scripts are more concise than customized test drivers, while achieving the same level of control-flow coverage, and much better coverage of combinations of parameter and state values. Also, APE scripts yield greater consistency in the way unit tests are written, improving comprehensibility and reducing maintenance costs.

Acknowledgements

This research was funded by formalWARE, a two-year industry/university collaborative research project sponsored by the British Columbia Advanced Systems Institute, Hughes Aircraft of Canada Limited, and MacDonald Dettwiler Associates. Part of the work was carried out during the internship of one of the authors from May 1 1997 to December 6 1997 at the International Airspace Management Systems Division of Hughes Aircraft of Canada Limited in Richmond, Vancouver.

References

[1] S. Barbey. Testing Ada 95 object-oriented programs. In M. Toussaint, editor, *Ada in Europe: second International Eurospace Ada Europe Symp.*, pages 406–418. Springer Verlag, 1995. Lecture Notes in Computer Science 1031.

[2] S. Barbey and D. Buchs. Testing Ada abstract data types using formal specifications. In M. Toussaint, editor, *Ada in Europe 1994*, pages 76–89. Springer Verlag, 1994. Lecture Notes in Computer Science 887.

[3] R.V. Binder. Testing object-oriented software: A survey. *Software Testing, Verification, and Reliability*, 6:125–252, 1996.

[4] S.P. Fiedler. Object-oriented unit testing. *Hewlett-Packard Journal*, pages 69–74, April 1989.

[5] P.G. Frankl and R.K. Doong. The ASTOOT approach to testing object-oriented programs. *ACM Trans. on Software Engineering Methodology*, 3(2):101–130, 1994.

[6] M.J. Gallagher and V.L. Narasimhan. A software system for the generation of test data for Ada programs. *Microprocessing and Microprogramming*, 38:637–644, 1993.

[7] J. Gannon, P. McMullin, and R. Hamlet. Data-abstraction implementation, specification and testing. *ACM Trans. Program Lang. Syst.*, 3(3):211–223, July 1981.

[8] D.M. Hoffman. A CASE study in module testing. In *Proc. Conf. Software Maintenance*, pages 100–105. IEEE Computer Society, October 1989.

[9] D.M. Hoffman and P.A. Strooper. Automated module testing in Prolog. *IEEE Trans. Soft. Eng.*, 17(9):933–942, September 1991.

[10] D.M. Hoffman and P.A. Strooper. *Software Design, Automated Testing, and Maintenance: A Practical Approach.* International Thomson Computer Press, 1995.

[11] D.M. Hoffman and P.A. Strooper. The testgraphs methodology—automated testing of collection classes. *Journal of Object-Oriented Programming*, 8(7):35–41, 1995.

[12] D.M. Hoffman and P.A. Strooper. ClassBench: A methodology and framework for automated class testing. *Software: Practice and Experience*, 27(5):573–597, 1997.

[13] AdaTEST the solution for testing ada software. http://www.iplbath.com/.

[14] G. Murphy, P. Townsend, and P.S. Wong. Experiences with cluster and class testing. *Commun. ACM*, 37(9):39–47, 1994.

[15] D.J. Panzl. A language for specifying software tests. In *Proc. AFIPS Natl. Comp. Conf.*, pages 609–619. AFIPS, 1978.

[16] S.R. Waterman. Techniques for testing Ada 95. In K. Hardy and J. Briggs, editors, *Reliable Software Technologies—Ada-Europe '97*, pages 278–291. Springer Verlag, 1997.

Appendix

```
PackagePrefix
   "Map"
ExceptionPrefix
   "Baty_System_Exceptions"
SubPrograms
   <destroy,bind,unbind,bound,view_of,iterate,iterator_with_state_generic>
Exceptions
   <Wild_And_Fatal_Error,Serious_But_Not_Fatal_Error>
GlobalCode
{%
with Baty_System_Exceptions;
with Abst_Names;         -- For Map Subsystem Id
with Baty_System_Types;         -- For Hash Function
with Tms;            -- For testmate
%}
LocalCode
{%
   function Hash (Val : Integer) return Baty_System_Types.Hash_Value;
   K,tmp_sum1,tmp_sum2 : Integer;
   loadsize:array(1..5) of Integer :=(0,1,2,5,500);
   Max_Map_Size : Baty_System_Types.Initial_Maximum := 10;

   package Map is
      new Abst_Simple_Map_Generic (Domain => Integer,
         Range_Of_Map => Integer,
         Default_Range => K,
         The_Name => "The_Map",
         The_Number_Of_Buckets => Max_Map_Size,
         Hash => Hash,
         The_Subsystem_Id => Abst_Names.Subsystem_Id);

   Integer_Map : Map.Object_Ref;

   function Hash(val: Integer) return Baty_System_Types.Hash_Value is
   i: integer;
   begin
      i := val rem 10;
      if i = 0 then
         i := 10;
      end if;
      return Baty_System_Types.Hash_Value(i);
   end hash;


   procedure load(n: in Integer) is
   begin
      Map.Destroy(Integer_Map);
      for i in  1..n loop
         Map.Bind(i,2*i,Integer_Map);
      end loop;
   end load;

   function exp_sum(n: in integer) return integer is
```

```
   s: integer := 0;
   begin
      for i in 1..n loop
         s := s+i;
      end loop;
      return 2*s;
   end exp_sum;

   procedure Process1(The_Domain: Integer;
            The_Range:Integer;
            Continue: in out Boolean) is
   begin
      tmp_sum1 := tmp_sum1 + The_Range;
   end Process1;

   procedure Process2(The_Domain: Integer;
             The_Range: Integer;
            The_State: in out Integer;
            Continue: out Boolean) is
   begin
      tmp_sum2 := tmp_sum2 + The_Range;
      Continue := True;
   end Process2;

   package p1 is new Map.Iterator_Generic(Process => Process1);
   package p2 is new Map.Iterator_With_State_Generic(Process => Process2,State =>
Integer);
%}
cases

{%
for i in  1..loadsize'last loop
   load(loadsize(i));
   for j in  1..loadsize(i) loop
%}
      <unbind(loadsize(i)+1,integer_map),Serious_But_Not_Fatal_Error,dc,dc,dc>
      <{%k :=
map.view_of(integer_map,loadsize(i)+1);%},Serious_But_Not_Fatal_Error,dc,dc,dc>
      <bind(j,j,integer_map),Serious_But_Not_Fatal_Error,dc,dc,dc>

{%
   end loop;
end loop;
for i in  1..loadsize'last loop
%}

   <load(loadsize(i)).destroy(integer_map).load(loadsize(i)),noexc,dc,dc,dc>
{%
   tmp_sum1 := 0;
   tmp_sum2 := 0;
   p1.iterate(over_the_map => Integer_map);
   p2.iterate(over_the_map => Integer_Map,With_The_State => k);
%}
   <, noexc,tmp_sum1,exp_sum(loadsize(i)),integer>
   <, noexc,tmp_sum2,exp_sum(loadsize(i)),integer>
```

```
{%
   for j in reverse  1..1000 loop
      if j <= loadsize(i) then
%}
         <,noexc,bound(j,integer_map),true,boolean>
         <{%k:=map.view_of(integer_map,j);%},noexc,k,2*j,integer>
         <unbind(j,integer_map),noexc,bound(j,integer_map),false,boolean>
{%
      else
%}
         <, noexc,bound(j,integer_map),false,boolean>
{%
      end if;
   end loop;
end loop;
%}
<load(100).unbind(10,integer_map),noexc,dc,dc,dc>
```