Daistish: Systematic Algebraic Testing for OO Programs in the Presence of Side-effects

Merlin Hughes and David Stotts
Department of Computer Science
University of North Carolina
Chapel Hill, NC 27599-3175

Abstract

Daistish is a tool that performs systematic algebraic testing similar to Gannon's DAISTS tool [2]. However, Daistish creates effective test drivers for programs in languages that use side effects to implement ADTs; this includes C++ and most other object-oriented languages. The functional approach of DAISTS does not apply directly in these cases. The approach in our work is most similar to the ASTOOT system of Doong and Frankl [1]; Daistish differs from ASTOOT by using Guttag-style algebraic specs (functional notation), by allowing aliasing of type names to tailor the application of parameters in test cases, and by retaining the abilities of DAISTS to compose new test points from existing ones. Daistish is a Perl script, and is compact and practical to apply. We describe the implementation and our experiments in both Eiffel and C++. Our work has concentrated on solving the semantics-specific issues of correctly duplicating objects for comparison; we have not worked on methods for selecting specific test cases.

Daistish consists of a perl script and supporting documentation. The current distribution can be obtained via WWW at URL

`http://www.cs.unc.edu/~stotts/Daistish/`

1 Related work

We have based our work on Gannon's DAISTS system [2]. DAISTS used Guttag-style algebraic specifications [6, 4, 3] as a test oracle for systematic testing of ADT implementations in a functional language. A functional language was used because algebraic specs are usually written in a functional notation. The basic procedure is to select appropriate data points (values for the parameters to the operations called in the axioms), compute the right and left sides of an axiom separately, and then compare the results. A correct implementation should produce values for each sice that are equivalent.

The DAISTS method is not easily applied to object-oriented languages, as they tend to operate not by function applications, but rather as procedures (methods) that alter the persistent state of the object by side-effect. To be compatible with the functional notation of traditional algebraic specs, we think of such procedures as being passed the object state *by-reference*; in the absence of concurrency (so for C++ and Eiffel) we can model this as a function that receives the object state as parameter and returns an object state as value. Our specs, then, use functional notation but indicate in many of the functions that the parameters and return values are handled *by-reference*.

The project most similar to Daistish is the ASTOOT system by Doong and Frankl [1]. ASTOOT is also based on DAISTS, and allows this form of testing for object-oriented languages (they specifically reported on an Eiffel version). Rather than use the traditional functional notation of algebraic specs, ASTOOT uses a spec language called LOBAS that is similar in notation to the trace specs of Parnas [9, 7]. LOBAS includes the useful ability to specify *inequal-*

ity between left and right sides of an axiom, something we have not worked on in Daistish. We will compare the details of Daistish and ASTOOT after explaining the structure and functioning of Daistish.

Another body of related work is the two-level specifications in Larch [5, 11, 12]. Specs in Larch have a part written as algebraic axioms, in functional notation; a second part describes semantic-specific details for a particular programming language. While a considerable body of work has been done with Larch for verification, we are not aware of any testing work using Larch specs.

2 Structure of Daistish

Daistish is a Perl script which processes a formal specification of an ADT, along with the code for an object implementing the ADT, to produce a test driver. Our first prototype was developed for programs written in Eiffel [8]. We have developed a second implementation for a reasonable subset of C++, and we are continuing to refine and experiment with the C++ version. The remainder of this section explains the details of the Eiffel version of Daistish; the C++ version is discussed following in section 4.

2.1 Specification format

The information processed by Daistish contains Guttag-style algebraic specifications of abstract data types. Figure 1 illustrates the format with the specification for a "stack" ADT. There are four main sections: *aliases*, *signatures*, *axioms*, and *vectors*. The *aliases* section contains alternate names for types, so the user can segregate test points and avoid language-specific pitfalls. The *signatures* section contains information about the methods of the object, and allows the user to indicate which parameters are used by-reference vs. by-value; it is here that we indicate procedures as functions that operate on the object state "by reference" or by side-effect. The *axioms* section contains definitions of comparisons for sequences of object method invocations. The *vectors* section contains definitions of specific values for the types that appear as arguments to the methods invoked in the axioms; as in DAISTS, we allow new test vectors to be named and to be built from previous vectors.

These specification sections can be in one file or in several. One use for this is to limit the explosion of test cases that results from crossing all test points of a certain type with the parameters of that type in the axioms. For example, one can put the aliases and signatures in one file, and then create several other files each containing a group of related axioms and a group of applicable test points. Another method of controlling explosion of cases (if that is desirable) is discussed in the *aliases* section following.

Aliases

The *alias* section assigns alternate names to types and functions, and provides additional flexibility, allowing a user to distinguish between two different abstract uses for a single type. The parser is not equipped to cope with function overloading so it is necessary to assign unique aliases to different instances of overloaded function names within definition files. Similarly, all type and function names must be simple identifiers; aliases must be provided and used for compound names such as '`new obj`' and '`template [class]`'.

As partial compensation for these burdens, aliasing allows pseudo-types to be used within axioms. It may be desirable to partition the test vectors of a particular type in order to reduce the number of tests performed (to limit combinatorial explosion). This can be achieved by using different aliases of the same type in axioms. Consider, for example, an ADT "finiteStack (int)" in which an integer is used to establish the maximum size of a stack, and integers are also used as the elements of the stack. With aliases two types can be created (say, *sizeInt* and *eltInt*) so that integer values that are intended to be elements are not applied to axioms as size parameters, and *vice versa*.

The strong typing of Daistish then requires a type conversion function for comparison of comparable types. This conversion function can be aliased to the null operation '`()`'. A similar type conversion function is required if a user type corresponds directly to a simple type of the underlying language and assignment from the simple type is desired. An example of this appears in figure 1:

```
intToId  ()
```

Signatures

The signatures file describes the signature of functions used in axioms and test vectors. Signatures must also be provided for the type conversion functions described above.

Signatures are partially complicated as a result of the purely functional definition files be-

```
# Aliases
  -- alias      real name
     stack      stack [id]
     intToId    ()
     printBool  io.putbool
     printId    print
     printStack print
# Signatures
  -- type       name       arguments
     {stack}    create
     {stack}    push       ({stack}, id)
     {stack}    pop        ({stack})
     id         top        ({stack})
     boolean    isEmpty    ({stack})
     id         intToId    (int)
     id         nullId
  -- user-defined for testing
     boolean    equals     ({*}, *)
     boolean    notEquals  ({*}, *)
     boolean    bigger     ({*}, *)
     {}         printBool  (boolean)
     {}         printId    ({id})
     {}         printStack ({stack})
# Axioms
  -- name       rule
     PopNew     equals(pop(create),create)
     PopPush    equals(pop(push(stack,element)),stack)
     TopNew     top(create) = nullId
     TopPush    top(push(stack,element)) = element
     EmptyNew   isEmpty(create) = true
     EmptyPush  isEmpty(push(stack,element)) = false
     PushPop1   notEquals( pop(push(stack,element)), push(pop(stack),element) )
     grow1      bigger( push(push(stack,element1),element2), push(stack,element1) )
# Vectors
  -- name       value
     id0        intToId(0)
     id1        intToId(1)
     id2        intToId(2)
     stack0     create
     stack1     push(stack0,id0)
     stack2     push(stack1,intToId(3))
```

Figure 1: Example ADT specification in Daistish

ing incommensurable with side effect functions. '`object.normalize (...)`' is an example of this; the object itself is an implicit parameter and result of the operation '`object := normalize (object, ...)`', and this must be explicitly specified in the signature. Permitting non-functional specification would be possible with a more extensive implementation.

Functions which are declared to return no value are assumed to be *display* functions for variables of the type given as parameter. If present, these functions are invoked to report the values returned by the left and right hand side of an axiom should the axiom fail. This helps to identify the cause, or at least to demonstrate the effect, of an error.

Axioms

Axioms are named, functionally-specified expressions that evaluate to '`true`' if the axiom is satisfied by the parameters. A limited number of infix expressions are available. Daistish automatically determines the types and name of free variables in the expression and assigns them values from the test vectors before evaluation.

Violations of Eiffel preconditions during evaluation of either side of an axiom causes that evaluation of the axiom to be ignored; it is reported as an aborted attempt rather than a test case failure. Aborted attempts usually result from invalid parameter selections and can often be eliminated by use of type aliases or by partitioning the test cases into separate spec files. Failure of one side of an axiom and not the other indicates a possible inconsistency. Failure of the root operation of the axiom (i.e, *equals*) is also an aborted attempt. All other exceptions indicate failure of the axiom and are considered to indicate errors in either the axioms or the implementation.

Test Vectors

Test vectors are named, functionally-expressed sample instantiations of types used by the axioms. At least one test vector must be supplied for each type used by a set of axioms.

A new instantiation of each test vector is created for each reference to the vector during each axiom evaluation. In cases where it is desired that references to the same vector are to the same instantiation, such as code which tests equality of reference, a test vector can be specified to be instantiated only once for each side of an axiom per evaluation.

2.2 Specifics of the notation

Several points about the notation shown in figure 1 need further explanation. First, as in the ASTOOT system, we can handle *not equal* comparisons. Rather than use tags in test cases, the capability in Daistish comes from the axiom directly. Comparisons for equality require a user-written function, and the user names this function arbitrarily. Usually, the spec writer will choose the name "equals", as we have done in this example:

```
PopNew  equals(pop(create),create)
```

However, there are times when other functions may need to be invoked. For example, Eiffel has a built-in comparison function called "equal" supplied with each class. This is sometimes (though not usually) sufficient for correct equality comparison of two objects. If it will work, the user can employ it by simply naming it in an axiom:

```
PopNew  equal(pop(create),create)
```

Since we allow any function to be the top invocation of the axiom tree, we can also write a function such as

```
notEquals (s1,s2)
   { return not(equals(s1,s2)) }
```

and then write an axiom like this:

```
trans1
   notEquals( pop(push(stack,elt)),
              push(pop(stack),elt) )
```

If an axiom is written using the "=" symbol, Daistish will generate code employing the algebraic operation appropriate for the types being compared.

The last axiom in figure 1 illustrates use of a general user-defined boolean function for comparison:

```
grow1
   bigger( push(push(stack,elt1),elt2),
           push(stack,elt1) )
```

In this case, we assume the function "bigger" determines if if its first argument (a stack object) is larger (in some appropriate sense) than its second stack argument. As with "equals" and "notEquals" the Daistish test driver will report an error if this boolean function evaluates to '`false`'.

Daistish allows users to get several pieces of information when an axiom test fails. The test generator will output the axiom name that failed, and the names of the data points used as parameters. It will also invoke user-supplied print functions (if they are given) to output the left-side and right-side values for the axiom. The *signatures* section of the spec contains the names and typing information for as many print functions as the user chooses to write.

Each print function will take one parameter; it will return no value and will not alter the state of the object. The parameter may be either by-reference or by-value, and the "{}" convention is used to indicate the difference. For the print function of an object, the spec will look like:

```
{}  printStack ({stack})
```

meaning it operates on a stack object, but takes it by-reference; it results in code like this in the test driver:

```
val.printStack ();
```

where "val" is an object of class "stack". For a simple type, which can be output by a function in the Eiffel io library, the spec would be something like:

```
{}  printBool (boolean)
```

indicating a call-by-value. We then alias *printInt* to

```
printBool  io.putbool
```

resulting in code like this in the test driver:

```
io.putbool (val);
```

This is a neat and fairly general approach.

2.3 Implementation notes

- Daistish scans all of the input files and builds parse trees for each axiom and test vector. Function signatures allow the types of all free variables to be determined. Code is produced to instantiate each test vector and evaluate each axiom. The axioms are then called with each possible valid combination of parameters available from instantiations of the test vectors. Statistics are collected for each axiom and summarized at completion.
- The number of axiom evaluations rises with the product of the number of test vectors of each free variable in an axiom. Care should be exercised not to supply too many test vectors of any type, and to partition common types into pseudo types where possible.
- In an Eiffel implementation, precondition failure is dealt with using '`recover`' clauses.
- Addition '`+`', subtraction '`-`' and multiplication '`*`' are supported on integers. Equality '`=`' (of reference) may be tested against any two items of the same type. Logical operations and '`&`' and or '`|`' are supported. Infix operators have equal precedence; the operator is applied to the item immediately to the left and the expression to the right; the right hand expression may involve operators; thus, '`A + B = C`' is incorrect for a triplet of integers; it is evaluated as '`A + (B = C)`'. Rather, '`(A + B) = C`' and '`C = A + B`' are correct. Brackets should be used for clarity.
- If-then-else clauses are effected with the conditional operator as in C: '`?` *`if-boolean`* `:` *`then-clause`* `;` *`else-clause`*'.

3 Object Cloning

It is not reasonable to re-instantiate a test vector for each occurrence of it as a free variable in an evaluation of an axiom; this would preclude the ability to use equality of reference in, for example, an expression like '`has (push (aSet, anEl0), anEl1)`' (where the '`has`' operation tests something like set membership). Neither is it reasonable to instantiate it only once per axiom evaluation because the side effects of evaluating one side of the axiom would mutate the objects involved in evaluation of the other side, causing an incorrect initial state for evaluation. The refinement of instantiating each test vector once per side of an axiom may cause complications if several free variables reference a single test vector and the variables are modified by side effect; the result would depend on the order of evaluation of parameters within the axiom.

This problem would not occur in a purely functional language. Where objects and side effects are allowed however, the problem must be dealt with in some fashion; in Daistish, test vectors may be declared to be either instantiated at every occurrence or once per side per axiom evaluation. Where equality of reference is never used, the former is safe and usually correct; where equality of reference or other complex behaviour is desired, the latter must be applied, with caution. When a test vector is declared to have the latter behaviour, it may be useful to declare a duplicate with the same initialization in order to evaluate the effect of different variables having the same value, but different references. This flexibility covers all situations with which the authors have dealt to date.

To this end, it is required that all axioms have a clear left and right hand sides. An axiom of the form '`has (push (aSet, anEl0), anEl0)`' is not valid; instead, the expression must be written as '`has (push (aSet, anEl0), anEl0) = true`' which has two children at the root of the parse tree, each representing a distinct side of the axiom.

4 The C++ version of Daistish

A C++ back-end for Daistish has been implemented. Clean handling of precondition violations requires exceptions; these were not supported by the compiler at the time of writing and thus crudely implemented with a global exception flag.

The current implementation also does not support all combinations of passing objects by value and reference. All parameters are passed by value; modifiable parameters must be declared reference arguments rather than pointers. Complex returned values (non-basic types) must be returned by reference rather than by value.

Removing these restrictions requires introducing the notion of pointers, the different handling of array and non-array parameters and the different means of initializing simple variables, arrays, structures and objects. This enhancement is planned for the near future.

We will be testing the C++ version in a formal methods class at UNC in the fall of 1995. The results from this evaluation will be available for conference presentation.

5 Summary of Daistish features

As stated earlier, Daistish and ASTOOT are very similar in the problem being attacked and the approach to a solution. We summarize here the differences between the two systems:

- Both allow equality and inequality comparisons; Daistish also allows general user-defined boolean functions to be used for comparisons.
- Daistish has been implemented for C++ in addition to Eiffel.
- Daistish allows compositional construction of test data.
- ASTOOT generates some test points automatically; Daistish provides no support for test point generation, requiring the user to do this explicitly.
- Daistish uses the traditional functional notation of algebraic specs; ASTOOT employs an operation sequence notation similar to that of Parnas' trace specs.
- Daistish has a typing system that allows aliasing of type names for (among other things) control of allocation of test points to axiom parameters.
- Daistish formal specs can be partitioned into several different files, allowing further control over explosion of ineffective test cases.
- In reporting test failures, Daistish will invoke user-defined print functions (if provided) to report the left- and right-side values of the failed axiom.

6 Evidence of effectiveness

We have not undertaken a formal evaluation of testing with Daistish, but we have some anecdotal evidence suggesting that it can profitably augment the testing methods commonly found in industry.

We first experimented with Daistish in the fall of 1994, in a graduate class on formal methods at UNC Chapel Hill. Students worked in teams of three; each student had at least an undergraduate level of programming expertise. Each team was required to write algebraic specs for a Petri net object (PNET), which formed the basis of a simplified hypermedia engine along the lines of Trellis [10]. The teams produced specs containing on the order of 60-80 axioms to fully express the behavior of the type. A portion of a spec file for the Petri net type is shown in figures 2 and 3. This example is more complicated and realistic than the *stack* from figure 1. Many of the axioms and test points have been excluded for brevity.

```
# Aliases
    --  alias           real name
        pnet            pneteq
        place           integer
        trans           integer
        queue           fixed_queue[integer]
        create_net      create
        create_place    ()
        create_trans    ()
        create_queue    create

# Signatures
    --  type            name            arguments
        {pnet}          create_net      (integer)
        {pnet}          addp            ({pnet},place)
        {pnet}          addt            ({pnet},trans)
        {pnet}          addpt           ({pnet},place,trans)
        {pnet}          addtp           ({pnet},trans,place)
        {pnet}          mark            ({pnet},place)
        integer         ntok            ({pnet},place)
        boolean         isin            ({pnet},place,trans)
        boolean         isout           ({pnet},place,trans)
        boolean         isp             ({pnet},place)
        boolean         ist             ({pnet},trans)
        boolean         enabled         ({pnet},trans)
        {pnet}          fire            ({pnet},trans)
        {pnet}          delpt           ({pnet},place,trans)
        {pnet}          unmark          ({pnet},place)
        {pnet}          putPcont        ({pnet},string,place)
        {pnet}          putPlabel       ({pnet},string,place)
        {pnet}          putTlabel       ({pnet},string,trans)
        string          getPcont        ({pnet},place)
        string          getPlabel       ({pnet},place)
        string          getTlabel       ({pnet},trans)
        queue           getMarked       ({pnet})
        queue           getEnabled      ({pnet},place)
        {queue}         create_queue    (integer)
   -- other operations in the mix
        boolean         equals          ({*}, *)
        boolean         equal           ({*}, *)
   -- necessary from aliases
        place           create_place    (integer)
        trans           create_trans    (integer)
```

Figure 2: More complicated ADT specification: Petri net, part 1

```
# Axioms
    ntok_1      ntok(create_net(15),pi) = 0
    ntok_2      ntok(addp(N,pj),pi) = ? equal(pi,pj) & (equal(isp(N,pj),false))
                                      : 0
                                      ; ntok(N,pi)
    ntok_3      ntok(addt(N,ti),pj) = ntok(N,pj)
    ntok_4      ntok(addpt(N,pj,tk),pi) = ntok(N,pi)
    ntok_5      ntok(addtp(N,tj,pk),pi) = ntok(N,pi)
    ntok_6      ntok(mark(N,pj),pi) = ? equal(pi,pj) : ntok(N,pi)+1 ; ntok(N,pi)
    ntok_7      ntok(putPcont(N,s,pj),pi) = ntok(N,pi)
    ntok_8      ntok(putPlabel(N,s,pj),pi) = ntok(N,pi)
    ntok_9      ntok(putTlabel(N,s,t),p) = ntok(N,p)
    enabled_1   enabled (create_net(15),t) = false
    enabled_2   enabled (addp(N,p),t) = enabled(N,t)
    enabled_3   enabled (addt(N,tj),ti) =
                    (equal(ti,tj)&equal(ist(N,tj),false)) | enabled(N,ti)
    fire_1      equals ( fire(create_net(15),ti) , create_net(15) )
    delpt_5     equals ( delpt(addtp(N,ti,pi),p,t) , addtp(delpt(N,p,t),ti,pi) )
    getEnabled_1    equal ( getEnabled(create_net(15),p) , create_queue(15) )
    getEnabled_2    equal ( getEnabled(addp(N,p),pi) , getEnabled(N,pi) )
    getEnabled_3    equal ( getEnabled(addt(N,t),p) , getEnabled(N,p) )

# Vectors
    pid_a     create_place(1)
    pid_b     create_place(2)
    tid_a     create_trans(3)
    tid_b     create_trans(4)
    label_a   "fooDeeBar"
    label_b   "withyWindle"
    net_0     create_net(20)
    net_a     putPlabel ( addpt ( mark ( addt(addp(net_0,pid_a),tid_a)
                                       ,pid_a )
                               ,pid_a
                               ,tid_a )
                       ,"init_P_cont"
                       ,pid_a )
    net_b     putTlabel( addtp ( addpt ( addt(addp(net_a,pid_b),tid_b)
                                       ,pid_b
                                       ,tid_b )
                              ,tid_a
                              ,pid_b )
                      ,"init_T_label"
                      ,tid_a )
```

Figure 3: More complicated ADT specification: Petri net, part 2

System implementation was done in Eiffel. After implementation, and after informal unit testing, each team applied the Daistish tool. All teams reported finding numerous "errors" in their implementation. Many of these errors were actually incomplete or erroneous axioms, as one would expect for students learning the method of algebraic specs. However, each team found numerous cases of legitimate missing or incorrect behavior of the code; again, this was after unit testing. These errors were mostly examples where the systematic application of test cases uncovered data situations the programmers had not anticipated or accounted for.

We developed the C++ version of the Daistish Perl script in spring 1995, after the class, and made it available this spring to the students that had taken the formal methods class. Two students who work for local companies in the Research Triangle Park reported to me that they had begun to write algebraic specs and apply Daistish in their professional programming, thought it was in an informal capacity (i.e., not necessarily sanctioned by their companies). They reported back to me finding errors in some previously written code that they had considered thoroughly tested. At this time, we do not have anything more specific than these anecdotes about industrial applications. We are pursuing more formal studies with two companies participating in the Software Engineering Research Center (SERC, sponsored by NSF at Purdue, U. Florida, and Oregon) and hope to be able to report results from a well-designed evaluation in a later paper.

Acknowledgements

The authors gratefully acknowledge John Gannon and the referees for several helpful comments and suggestions for improving the effectiveness of Daistish.

References

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

[2] Gannon, J., McMullin, P., and Hamlet, R. Data-abstraction implementation, specification, and testing. *IEEE Transactions on Programming Languages and Systems 3*, 3 (July 1981), 211–223.

[3] Guttag, J. Notes on type abstraction (version 2). *IEEE Transactions on Software Engineering TR-SE 6*, 1 (Jan. 1980), 13–23.

[4] Guttag, J. V., and Horning, J. J. The algebraic specification of abstract data types. *Acta Informatica 10* (1978), 27–52.

[5] Guttag, J. V., Horning, J. J., and Wing, J. M. The Larch family of specification languages. *IEEE Software 2*, 5 (September 1985), 24–36.

[6] Guttag, J. V., Horowitz, E., and Musser, D. R. Abstract data types and software validation. *Communications of the ACM 21* (Dec. 1978), 1048–1063.

[7] Hoffman, D., and Snodgrass, R. Trace specifications: Methodology and models. *IEEE Transactions on Software Engineering 14*, 9 (Sept. 1988), 1243–1252.

[8] Meyer, B. *Eiffel: The Language*. Prentice Hall, 1992.

[9] Parnas, D. L., and Wang, Y. The trace assertion method of module interface specification. Technical Report 89-261, Queen's University, Kingston, Ontario, Oct. 1989.

[10] Stotts, P. D., and Furuta, R. Petri-net-based hypertext: Document structure with browsing semantics. *ACM Transactions on Information Systems 7*, 1 (Jan. 1989), 3–29.

[11] Wing, J. M. Writing Larch interface language specifications. *ACM Transactions on Programming Languages and Systems 9*, 1 (Jan. 1987), 1–24.

[12] Wing, J. M. Using Larch to specify avalon/c++ objects. *IEEE Transactions on Software Engineering 16*, 9 (September 1990), 1076–1088.