Daistish^{++}

a new and improved DAISTS-like axiom tester

Merlin P. D. Hughes

September 25, 1995

1 Execution

When daistish is applied to a set of axiom specification files, a C++ tester program is produced. When compiled and run, this program will test the adherence of an implementation of a data type against axia describing its characteristics. If the code is executed with a '`-verbose`' command line option, detailed reports are given for each axiom failure.

2 Syntax

`daistish [<`*`options`*`>] <`*`axiom files`*`>`, where options are:

- `-o <`*`output file`*`>` — specifies the name of the C++ test program to be created. If the filename has a '`.c`' suffix, the header file will have a '`.h`' suffix. Similarly will a '`.C`' file have a '`.H`' header, and a '`.cc`' file '`.hh`'.
- `-i <`*`include file`*`>` — specifies a header file to include. Typically this will include the header file for the class being tested.
- `-a <`*`alias file`*`>` — specifies an alias file to be processed.
- `-s <`*`signature file`*`>` — specifies a signature file to be processed.
- `-v <`*`test vector file`*`>` — specifies a test vector file to be processed.

Daistish automatically checks for a '`.daistishrc`' resource file which may contain any of the above options. If an option is given on the command line, any other options of that type given in the resource file are ignored.

```
-o Tester.cc
-a Alii.d -v Vectors.d -s Signatures.d
-i Petrinet.hh -i IdSet.hh
```

Figure 1: Example `.daistishrc` file.

2.1 Alias File

<alias name> *<real name>*

The alias file allows aliases to be assigned to type and function names in the axiom specifications. There are three reasons to to do this; firstly, the simple parser requires all type and function names to be simple identifiers like C variable names; secondly, pseudo types can be used to reduce the explosion of combinations in test vector space; these pseudo types are then all aliased to be the same 'real' type. This latter problem is discussed in more detail later (§2.3.2). Aliases can also be used to represent pseudo constants which have been declared as functions without parameter lists.

The alias name must be a simple identifier, the 'real' name is taken to be the rest of the text *on the same line* as the alias name; this may be blank. Whenever the alias name occurs as a type or function name in the generated code, it will be replaced by the 'real' name.

```
Id            Id *
NodeId        Id *
TransId       Id *
Attr          Attr *

newId         new Id
newAttr       new Attr
nullAttr      new Attr

cout          cout <<
```

Figure 2: Example alias file.

In the example alias file (figure 2), all class names are aliased to be references to the class type. This is done because the particular piece of code being tested accesses all class variables by reference; the constructor (`new`) returns a reference to a new instance and all classes used as parameters are passed by reference. Type and function names in daistish must be simple identifiers so the contruct '*type* `*`' cannot be used as a type name in the axiom specifications.

Classes may also be used by value. The usual constructor '`new`' must be deferenced if it is to be used as the constructor; this can be achieved through a function aliased to the C derefence operator '`*`'. If a class is to be used by value, two constraints are made upon it; assignment between instances of the class must be supported and the class must be instantiable without parameters. This latter requirement is necessary to support temporary variables which will be declared as '*type variable*;'.

The aliasing of the '`new`' and '`cout`' functions is another symptom of the simple identifier constraint; compound function names cannot be used in axiom specifications so a simple alias is provided.

The alias '`nullAttr`' is a pseudo constant; it occurs in the axiom specifications to indicate the value of a null '`Attr`'; in this code implementation, that is equivalent to a new instance of the type.

2.2 Signature File

<return type> <function name>
<return type> <function name> (<type list>)

Signature files specify the signatures of functions used in axiom specifications. There is a difference between specifying a function without a type list and specifying a function with an empty type list; the former generates code without parentheses following the function name, the latter generates code with parentheses but no parameters. The former can be used to indicate the type of system constants such as 'MAXINT' or code-specific constants such as 'nullAttr'; the latter indicates a function taking no parameters.

```
bool equalIsh ({*}, *)

{} cout (int)
{} cout (Id)

Petrinet newPetrinet ()
{Petrinet} attrChg ({Petrinet}, Id, Attr, Value)
bool attrQue ({Petrinet}, Id, Attr)

Id newId (int)
Attr newAttr (string)
Attr nullAttr
```

Figure 3: Example signature file.

Daistish employs functional notation throughout, which is somewhat at odds with the object oriented paradigm. In effect however, a method of a class can be expressed as a pure function which takes the class as a parameter; this is the notation which is required by daistish: 'class.mutate (value)' is in effect, 'class = mutate (class, value)'; 'variable = class.method ()' is in effect 'variable = method (class)'.

Curly braces are used to indicate the location of this class parameter; they are placed around the parameter which represents the instance of which the function is a method. If a method is a mutator of a class, the return type is in effect the class itself; this is indicated by enclosing the return type in braces.

En anglais, if a signature has both the return type and a parameter enclosed in braces, it is a mutator of the embraced class. It is senseless for the embraced return type and parameter to be different. It is senseless to have an embraced return type but no embraced parameter. If just a parameter is enclosed in braces, the function is a method of that class.

In the example signature file (figure 3), 'attrChg' is a mutator of class 'Petrinet', 'attrQue' is a boolean-returning method. If an axiom includes the expression 'attrChg (net, id, attr, value)', code is generated with the form '`net.attrChg (id, attr, value)`' and the result is the variable '`net`'. The expression '`attrQue (net, id, attr)`' has the result '`net.attrQue (id, attr)`'.

Daistish performs strong type checking; it supports overloaded function names so a single function name may have multiple signatures as long as they are distinguishable. The wildcard '`*`' is a valid parameter type name and will accept any variable of any type.

2.2.1 Printing functions

A signature which has a return type of an empty pair of braces (*i.e.* nothing is returned and it is returned by side effect) signifies a function which will print a variable of the type of its one parameter. In the example signature file, '`cout`' (see also the alias file) is specified as being capable of printing integers and identifiers. The type may be enclosed in braces if the printing function is a method of the type.

When an axiom fails, the printing function (if it exists) is invoked for the failing lvalue and rvalue.

2.2.2 Known types and functions

Daistish knows the types '`int`', '`bool`', '`string`' and '`char`'.

Integers are specified by an optional negative sign followed by one or more digits; type '`bool`' is aliased to '`int`' and specified by '`true`' or '`false`', those being aliased to '`1`' and '`0`' respectively); strings (`char *`) are enclosed in double quotes, chars in single quotes.

```
bool == (*, *)            int + (int, int)
bool && (bool, bool)      int - (int, int)
bool ^^ (bool, bool)      int * (int, int)
bool != (*, *)            int / (int, int)
bool || (bool, bool)      int % (int, int)
```

Figure 4: Built-in infix functions.

Built in infix functions with their signatures are listed in figure 4. Evaluation of infix functions is *strictly* right to left; there is no precedence. '`a == b ? c : d`' will be interpreted as '`a == (b ? c : d)`'. Use brackets profusely. The only other built in function is '`bool not (bool)`'.

Conditional evaluation is performed with the C-like ternary operator: '<*if boolean*> ? <*then clause*> : <*else clause*>'.

It should be noted that logical OR and AND are not necessarily going to be evaluated as OR-ELSE and AND-THEN as they are in C; side-effect functions may require partial evaluation of both sides of the operator before it itself can be applied. If the alternative behaviour is desired, the ternary operator must be used.

2.3 Test Vector File

<vector name> <expression>

The test vector file specifies all the test vectors against which the axia will be tested. Test vectors must be given for the types of all free variables in the axia. The type of each vector is automatically determined from the expression.

The expression may include any known function, any built-in constant and any other test vector name; test vectors can be used before they are defined but recursive definitions are prohibited.

2.3.1 Single Evaluation Vectors

If the vector name is enclosed in braces, it is a "once" test vector. This means that it will only be evaluated once per side per axiom evaluation; further references to it during evaluation of the same side of the axiom will receive the previous reference value. This permits each side of an axiom to be evaluated with different free variables holding references to the same test vector object; ordinarily, a new instance of the test vector would be created for each free variable.

This mechanism is usually only required where object references are tested for equality however it should be noted that this reference equality is *not* preserved between the sides of an axiom so the equality check used between sides of an axiom (the root function of an axiom) must use some other formulation of equality. The example test vector (figure 5) uses an integer in the constructor of identifiers; this integer serves no purpose except for evaluation of the '`equalIsh`' function used by the axia; the integer value is used in place of the reference value for global equality checking. If homomorphism is a sufficient condition for equality and can be easily evaluated, the integer could be eliminated.

Code which uses this "once" mechanism should beware of side-effect functions which modify such variables; the order of evaluation of code is not specified so code which relies on the order of evaluation of parameters may not test correctly. The errors exhibited by such problems can be subtle.

The single evaluation nature of the test vectors is not carried across the two sides of an axiom; were it carried across and one side of the axiom modified (by side effect) a variable holding the test vector, the other side of the axiom would receive different initial variable bindings to the first because of the modification.

To be added: Test vectors may be evaluated once per execution of the code for all axia if surrounded by double braces. This should only be done if the test vector will never be modified by the code.

2.3.2 Controlling the Test Space Explosion

The number of tests performed by daistish rises with the product of the number of test vectors for each free variable in an axiom. It is sometimes useful to partition a type which is used in different rôles into pseudo-types corresponding to each rôle. For example, if an integer is used to specify the size of a set *and* the elements being placed into the set, one could create two new types;

```
{id_0}      newId (0)
{id_1}      newId (1)
nid_0       NodeId (id_0)
tid_1       TransId (id_1)
{attr_0}    newAttr ("Attr_0")
{val_0}     newValue ("Value_0")
net_0       attrChg (
              arcIns (
                tokenIns (
                  transIns (
                    nodeIns (newPetrinet (), nid_0),
                    tid_1),
                  nid_0),
                id_0, id_1),
              id_0, attr_0, val_0)
```

Figure 5: Example test vector file.

one corresponding to the set size and one to the elements, both aliased to be integers. If two test vectors of type '`size`' are provided (for example, by casting an integer ('`size (1)`')) and four of type '`element`', an axiom involving one free '`size`' variable and two free '`element`' variables will be evaluted $2 * 4 * 4 = 32$ times instead of the $6 * 6 * 6 = 216$ evaluations which would be required if an integer is used throughout. In the example test vector file, '`NodeId`' and '`TransId`' are a good example of this; their values are in fact taken from other "once" '`Id`' test vectors although this need not be the case.

2.4 Axiom File

<axiom name> <expression>

The axiom file consists of a series of named expressions; the root of each expression must be a boolean-returning binary function. The expression may include any known functions, constants and free variables. Free variables are identifiers corresponding to a particular instance of a variable occuring in the axiom. The type of each free variable name must be specified at least once in the axiom file; this is done by prefixing an occurence of the free variable with its type (*c.f.* figure 6; the type of '`nid1`', '`net`' and '`tid1`' is specified once each). A particular free variable name will always refer to one type so may not be re-specified with a different type later on.

To be added: Permit variable type specification outside of axia.

The *lvalue* (the left hand side of the axiom) will be compared with the *rvalue* using the root function for *each possible combination* of test vector assignments to the free variables of the axiom. If the result is '`false`', the axiom is considered to have failed. Upon failure, if testing is '`-verbose`', the free variable assignments will be indicated and if a suitable printing function has been specified, the *lvalue* and *rvalue* will be displayed.

To be added: Permit '`not`' at the root of an axiom.

```
nodedel_new

equalIsh (nodeDel (newPetrinet (), NodeId nid1),
          newPetrinet ())

nodedel_nodeins

equalIsh (nodeDel (nodeIns (Petrinet net, nid1),
                   NodeId nid2),
          (nid1 == nid2)
          ? net
          : nodeIns (nodeDel (net, nid2), nid1))

nodedel_transins

equalIsh (nodeDel (transIns (net, TransId tid1),
                   nid1),
          transIns (nodeDel (net, nid1), tid1))
```

Figure 6: Example axiom file.

Currently, due to the lack of exceptions, precondition violations is implemented by a global variable called '`DA_Abort`'. If an axiom is evaluated and this variable is non-zero, regardless of the axiom return value it will not be considered to have failed but to have aborted. The number of precondition violations is counted and reported at the end for each axiom; if preconditions are extremely stringent, there may be many more precondition violations than genuine axiom tests; this may be a concern and can sometimes be addressed by partitioning a variable type.

Figure 7 demonstrates the possible implementations of precondition and postcondition checking functions. If the precondition fails, the relevant daistish variable is asserted; and failure is returned; if the ensure clause fails, a warning is displayed.

```
#include <iostream.h>

extern int DA_Abort;

int require (char *requisite, int present)
{
  DA_Abort |= !present;

  return !present;
}

void ensure (char *ensuree, int ensured)
{
  if (!ensured)
    cerr << "Ensure: " << ensuree << endl;
}

void Petrinet::nodeDel (Id *id)
{
  if (require ("nodeDel::isanode",
               nodesTokens->maps (id)))
    return;
  ...
  ensure ("nodeDel::isnotanode",
          !nodesTokens->maps (id));
}
```

Figure 7: Example precondition function and its usage.