THE INTERLISP-D TESTING SYSTEM

The Interlisp-D testing system is an integrated system built for creating, managing and using a large set of programmed tests for testing the correctness and the performance of the Interlisp-D programming environment.

The system is consisted of three parts : The test driver, the data base management system, and a graphic control tool. In addition, there are various tools for helping the test builders in the process of creating new tests.

All parts of the system assumes the structure of a TEST which is a data type consists of several fields, of which the most important are the expression which has to be evaluated, and a predicate which takes the results of this evaluation and determines whether the test was a success or a failure (i.e. whether the actual result is the same as the expected result).

The test driver is in principal a function which gets an object of type TEST, performs the test, and return either success or failure plus some additional information. It includes facilities for monitoring the test execution, tracing and recording the testing process to enable reproducing tests, Remote Eval protocols to enable performing tests with two machines and more.

The data base management system works in two levels. In the low level, the "test cluster" level, the system manages and organizes the tests in the file system, enable retrieving tests through a cashing system, and allows concurrent access to test files using a simple locking scheme.

In the high level, the system enables each user to manipulate the database using its own VIEW of the system. This view is implemented through the CONCEPT SPACE which is a directed acyclic graph that will usually reflect the logic structure of the system as seen by the user.

The graphic control tool displays a concept space as a graph and allows the user to perform most of the Test system operations by selecting nodes from the graph.

The tools for building and manipulating tests include test inspector (and editor), a random generator which can generate random specified Lisp objects, Indirect reference to other tests in TEST fields for shrinking the space of the tests themselves and avoiding redundant work when creating tests which share some of their fields, and more .

In the next sections the different parts of the system will be described as well as the interaction between them.

The TEST data type

The TEST is the data type of test objects. Its structure reflects the various properties that tests have. It includes the following fields:

TestID : The tests are identified by an integer.

Input : This field contains an expression that, when evaluated, will generate the list of arguments on which the tested expression will be applied. There are several tools which help in creating this entry. The random generator helps in generating random objects with specified restrictions. The SYSTEMATIC operation helps in generating systematically all the combinations over finite ranges.

Expression: It can contain a function name, a lambda expression or arbitrary sexpression. In the first two cases it will be applied on the input.

Success Predicate: This field contains a lambda expression with two arguments - the ACTUAL input for the test, and the result of the evaluation. It returns one of the two atoms: Success or Failure. When performing tests with random input, some tricks may have to be used as demonstrated in the examples in the end.

Timeout :This is a lambda expression which gets the ACTUAL input as an argument, and produces an upper limit to the estimated time of evaluation.

EvalBefore and EvalAfter: expressions to be evaluated before and after the test execution. usually, before the test we may want to set the appropriate environment for the test (like loading certain files), and after the test we may want to clean up the environment (like deleting files which the test created).

Pretests: Contains a list of links to other tests. These links may influence the order of an execution of a set of tests. Currently there are two types of links. A STRONG link to other test means that whenever the current test is going to be executed, the pretest must be executed first. An example for such pretest may be tests for the tests themselves. If a test generates a few thousands combinations of some arguments, it may be useful to test first if the test itself works correctly by executing a simplified test which works only on one set of arguments, and check that the test outcome is reasonable. A WEAK link to other test means that whenever a SET of tests is being executed, and both the test and pretest are in this set, the pretest will be executed before the test (thus it defines a partial order on any set of tests). This link may be used in cases where there is logical order on the execution of tests - for example, it is reasonable to test opening a file before testing writing to a file.

The Test Driver

The test driver accept a test as its input and returns either success or failure. It will evaluate the input and the tested expression itself on a remote machine if requested, or on the local machine otherwise. All the process of the testing is recorded on a trace file, such that as much information as possible will be available if needed.

The driver evaluates the EVALBEFORE form, evaluates the input expression to generate the input for the tested expression, applies the tested expression on the generated input, applies the success predicate on the result and the generated input, and evaluates the EVALAFTER form. After some of the above stages the appropriate information is written on the trace file. The most important one is the input generated, especially in cases of random input.

All the evaluation done by the driver uses the Interlisp-D ERRORSET command, thus allowing evaluation that will not break under error condition. The error type may be used by the success predicate to determine if the result is a success or a failure. Thus one test for many arithmetic functions can be to supply them with non numeric arguments and to check that the error reported will the right one.

The evaluation of the tested expression is done as a separated process, such that the driver will be able to try to interrupt it in case where the time of execution is larger then the value of TIMEOUT field of the test. This interrupt will work only if the test execution process will release voluntarily the cpu (when waiting on I/O for example) since Interlisp-D uses non preemptive scheme for process scheduling.

Remote evaluation will not benefit us much in this type of problems. If the remote machine is in infinite loop for example, it will not listen to interrupt attempts as well. The advantages of using remote machine are two: If a long sequence of tests are executed, and the machine "freezes", a remote test will freeze the remote machine and the local machine will be able to call for help and resume operation (as soon as the remote machine does not respond for more then some estimated limit of time, the local machine sends messages to a preset distribution list and asking for human help). A second benefit of remote evaluation is when we need to evaluate the tested expression in a different environment than the Testing system resides. We will want the testing system to work in considerably nvironment (software release), while we are testing an experimental different environment.

The data base management system: the "test cluster" level.

The user can retrieve a test by calling the GetTest function. The low level of the dbms is responsible for performing the appropriate operations to retrieve the requested test. If the test is not already loaded it will be loaded from its file. There is a limit on the number of the tests that are loaded, and if these number is exceeded a replacement will take place and a test will be removed. The replacement policy is LRU (least recently used) and is implemented by moving each test being referenced to the front of the list of the loaded tests. Thus the last test in the list will be the one to be removed. The limit on the number of loaded tests is dynamically modified according to the amount of the available memory.

The Interlisp-D testing system is designed to work with several users uses it concurrently. There are no problems if the users were only retrieving tests from the data base. Problems may occur if two users modify the same part of the data base in the same time.

For such cases a locking scheme was integrated into the system. There is a special designated file which is the "gate" for the data base. Users can obtain write LOCKS on tests. The file contains the list of users with their locked tests. The basic locking function is ObtainDatabaseWriteLock(testnumber) which checks the LOCK file, and registers the tests that are not already locked . The user has the option of

automatically generated messages that will be sent to the locking users, inform them that somebody is waiting for their locked test s and request them to release them as soon as they are not needed.

Thus, either automatically or manually, whenever a user edits a test, he will first obtain LOCK on tests. The locking scheme will work only if the users will follow the rules and will not try to access tests not through the testing system.

The basic operation - ObtainDatabaseWriteLock is "atomic" in the sense that the LOCK file is opened for read and write throughout the execution of this procedure, and thus no other user will be able to open it. The time interval in which the file is opened is very short.

Another problem that may arise from concurrent access to the data base is test numbering. As mentioned above, each test has a unique integer as an ID. Thus there is a file which contains the last ID issued, and the procedures for creating new tests will access and update this file.

The "Concept Space" level.

What is the "thing" which is being tested by the test? it may be a specific low level system function, a library package, or a new representation scheme for integers. It is hard to find a common class to which all these entities belong. Thus the testing system assumes that it is some CONCEPT of the Interlisp-D system that is being tested.

While it is true that when a test is CREATED, its creator intent to test a specific concept, the test itself is not necessarily a test only for this concept. A test that was built for testing the READ function, may actually test also the NS communication protocols, the OPENFILE function etc.

For this reason the tested concept is not considered to be a part of the test itself. There is a separated knowledge strpace" which is the way that the user views the test cluster. A concept space is an acyclic directed graph of CONCEPTS. Each node of the graph is of type CONCEPT which has four fields: The concept name, the tests that tests this concept, the subconcepts and the superconcepts.

The main purpose of the concept space is to enable the user to group tests in a logical way and to perform operations on these sets of tests. The semantic of a concept node is : "the tests which tests this concept are the tests of the concept itself plus the tests that tests its subconcepts (recursively)".

Such a definition allows us to build concept spaces which view the tests from different points of view. We may have a concept named "Arithmetic system" with subconcepts "Integer arithmetic", "Flote arithmetic"and "Arithmetic functions". The "Arithmetic functions" will have as subconcepts, the concepts "IPLUS", "FPLUS", "PLUS" etc. "IPLUS" is also a subconcept of "Integer arithmetic", and "FPLUS" of "Float arithmetic". Thus, if a new representation for the integers was introduced to the system, we will test the "Integer Arithmetic" concept, while in other cases we may want to test the "Arithmetic Functions".

From the above example and from the more detailed example at the end, we can conclude that the organization of the test system should be very flexible, since there can be many parallel views into the same part of the system. We can also see why a tree structure would not be sufficient as a representation scheme. The Testing system supports the the co-existence of several concept spaces, and thus each user can build and use his own concept space(s) to reflect his view of the system.

THE CONCEPT SPACE BROWSER

Most of the operations of the Interlisp-D testing system are done through the "Concept Space Browser". The browser is a graphic tool which is applied on a concept space.

It has a few types of operations. Any operation that require a concept as an argument will get it by a selection from the displayed graph.

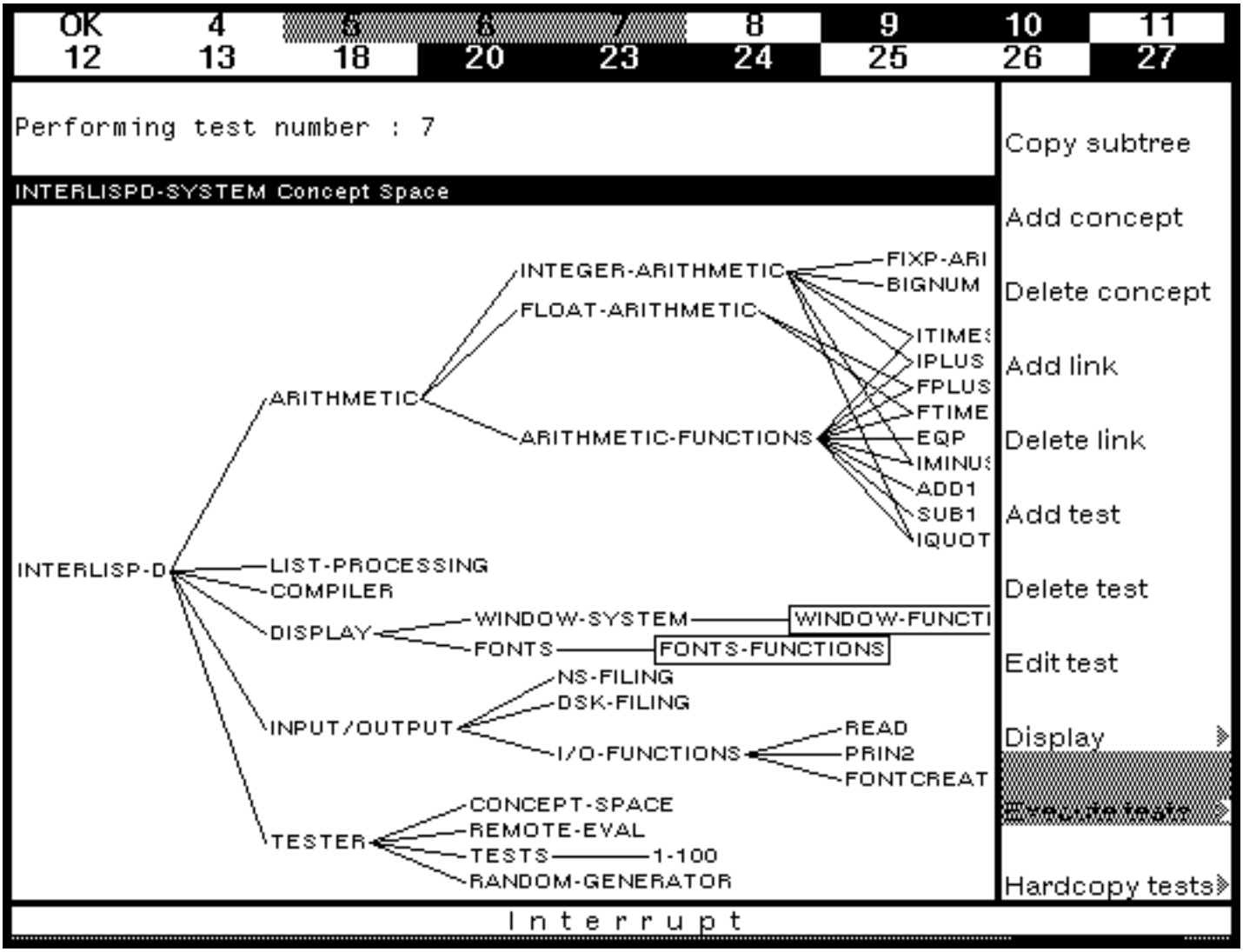
The first type of operations are operations for modifications of the concept space itself. There are commands to add new concept, to delete a concept, to ad and delete a link and to add and delete a test to a concept.

Second type of operations are data base operations. The user can edit a test selected from specific concept, can hardcopy all (or part of) the tests of a selected concept, and can request to lock all (or part of) the tests of a concept.

A third type are display op can be specified, a browser of a subgraph can be created, and the tests can be dre are commands to execute all (or part of) the tests of a certain concept, with different modes of execution .

The browser also allows to copy subgraphs between two displayed concept spaces, and to get all the tests of a node by a copy selection so that functions that are not available in the browser can use the concept space as well.

.



Test creation tools and misc tools

There are several tools that were created to help the tests builders in their task.

In each field of a test the user can write (& n) when n is a test number. This tells the system that to retrieve the value for this field it should refer to the same field in test n. This was done since many tests share values for some of their fields.

The test inspector is built on the top of the Interlisp-D inspector. The user can inspect and modify the various fields of a test. In addition he can call the inspector on indirect referenced tests.

There are functions which keep tracks of changes done to tests, and functions which stores modified tests.

The Random Generator is a very important tool for creating a test. It has many entries for different Lisp

objects which he can generate randomly. These set of possible entries will grow constantly as test builders will need more types of random objects.

The random generator function gets as an argument an object type and a list of modifiers. The test builder can specify in the input field of a test a call like (GenerateRandom 'LARGE-INTEGER), or (GenerateRandom 'WINDOW) or (GenerateRandom '(LIST-OF-ITEMS WINDOW 50 100)) to get a random list of length between 50 and 100 of windows.

From the experiments done with the Testing system it was clear that random tests are an important part of any testing and can discover bugs that would be hard to find otherwise.

Another tool is the SYSTEMATIC input generator. Many times we want to test a function if it works right with all the possible combinations of values of its arguments, or to find out whether an library package works with all possible settings for its flags. For such cases the test builder can specify in the input field that he wants a systematic test, and supply the expressions that produces the ranges of values to combine.

TEST EXAMPLES:

EXAMPLE 1

```
Test number 18
PRETESTS :NIL
TESTCOMMENT :(* * Generates systematicly all the pairs of all
                the "special" bignums
                (stored in TEST.BIGNUM-SPECIAL-NUMBERS in file
                        TEST-ARITHMETIC-UTILS)
                apply IPLUS on them and the IDIFFERENCE
                  and compare the results)
EVALAFTER :[LAMBDA (RES ARGS]
EVALBEFORE :(& 4)
TIMEOUT :NIL
TIMES :1
SUCCESSPREDICATE :[LAMBDA (RES ARGS)
                     (if (EQP (IDIFFERENCE RES (CADR ARGS))
                              (CAR ARGS))
                         then (QUOTE SUCCESS)
                       else (QUOTE FAILURE]
INPUT :(SYSTEMATIC TEST.BIGNUM-SPECIAL-NUMBERS
                   TEST.BIGNUM-SPECIAL-NUMBERS)
EVALEXPR :IPLUS
TESTID :18
```

EXAMPLE 2

```
Test number 14
PRETESTS :((WEAK 10))
TESTCOMMENT :(* * Creates a random window, reshapes it to a
                 sequence of random regions, and backwords to
                 the original shape ; Checks if the screen
                 bitmaps at the begining and end of operations
                 are equal)
EVALAFTER :[LAMBDA (RES ARGS)
             (CLOSEW (CAR ARGS]
EVALBEFORE :NIL
TIMEOUT :[LAMBDA (ARGS)
           1000]
TIMES :1
SUCCESSPREDICATE :[LAMBDA (RES ARGS)
                    (if (TEST.COMPARE-BITMAPS RES (CADR ARGS))
                        then (QUOTE SUCCESS)
                      else (QUOTE FAILURE]
INPUT :[LIST (TEST.GENERATE-RANDOM (QUOTE WINDOW))
             (BITMAPCOPY (SCREENBITMAP))
             (TEST.GENERATE-RANDOM (QUOTE (LIST-OF-ITEMS REGION
                                                         10 50]
EVALEXPR :[LAMBDA (WINDOW OLDSCREEN REGION-LIST)
            (PROG (OLD-WINDOW-REGION)
                  (SETQ OLD-WINDOW-REGION (WINDOWPROP
                      WINDOW
                      (QUOTE REGION)))
                  (for R in REGION-LIST
                     do (SHAPEW WINDOW R))
                  (for R in (APPEND (REVERSE REGION-LIST)
                                    (LIST OLD-WINDOW-REGION))
                     do (SHAPEW WINDOW R))
                  (RETURN (BITMAPCOPY (SCREENBITMAP]
TESTID :14
NIL
```

EXAMPLE 3

```
Test number 4
PRETESTS :((STRONG 11 23)
           (WEAK 7))
TESTCOMMENT :(* * Generates 1-30 BIGNUMS of the form 100...0,
                applies ITIMES on them,
                  and checks that the result is of the form
                100....00 with the right number of ZEROs)
EVALAFTER :NIL
EVALBEFORE :(LOAD? (QUOTE TEST-ARITHMETIC-UTILS))
TIMEOUT :[LAMBDA (ARGS)
           (IPLUS 10000 (ITIMES (LENGTH ARGS)
                                10000]
TIMES :10
SUCCESSPREDICATE :[LAMBDA (RES ARGS)
                    (PROG (SUM-OF-LENGTH UNPACKED-RESULT)
                          [SETQ SUM-OF-LENGTH
                            (for ARG in ARGS
                               sum (SUB1 (LENGTH (UNPACK ARG]
                          (SETQ UNPACKED-RESULT (UNPACK RES))
                          (if
                            [AND (EQP (ADD1 SUM-OF-LENGTH)
                                      (LENGTH UNPACKED-RESULT))
                                 (EQ (CAR UNPACKED-RESULT)
                                     (QUOTE 1))
                                 (for DIGIT
                                    in (CDR UNPACKED-RESULT)
                                    always (EQ DIGIT
                                               (QUOTE 0]
                              then (RETURN (QUOTE SUCCESS))
                             else (RETURN (QUOTE FAILURE]
INPUT :(TEST.GENERATE-RANDOM (QUOTE (LIST-OF-ITEMS
                                      POSITIVE-POWEROF10-BIGNUM 1
                                                        30)))
EVALEXPR :ITIMES
TESTID :4
```

EXAMPLE 4

```
Test number 16
PRETESTS :NIL
TESTCOMMENT :(* * This test create systematicly all
                combinations of arguments to FONTCREATE
                function,
                  and tests whether the only error is
                      "file not found")
EVALAFTER :[LAMBDA (RES ARGS)
             (if (NOT (TEST.ERRORP RES))
                 then (APPLY (QUOTE SETFONTDESCRIPTOR)
                             ARGS]
EVALBEFORE :NIL
TIMEOUT :NIL
TIMES :1
SUCCESSPREDICATE :[LAMBDA (RES ARGS)
                    (if (OR (NOT (TEST.ERRORP RES))
                            (EQP (CADR RES)
                                 17))
                        then (QUOTE SUCCESS)
                      else (QUOTE FAILURE]
INPUT :(SYSTEMATIC (QUOTE (GACHA NIL))
                   (QUOTE (-1 12 NIL))
                   [TEST.ALL-COMBINATIONS
                     (QUOTE ((BOLD MEDIUM LIGHT)
                               (ITALIC REGULAR)
                               (REGULAR COMPRESSED EXPANDED]
                   (QUOTE (0 90 NIL))
                   (QUOTE (DISPLAY PRESS NIL)))
EVALEXPR :FONTCREATE
TESTID :16
...
```