Intelligent Text Comparison in Software Validation

Inge Jechart, Ray Paquette, and Stefan Schmitz

The intelligent comparison tool (ICT) was developed to perform intelligent comparisons of text files. The purpose of the tool is to reduce the differences found when comparing two text files to only the meaningful or important ones. This tool contrasts with conventional differencing tools that find all literal differences. This tool was developed to help in the validation of software revisions where a large amount of text output is compared to a benchmark output.

Problem Description

Manual and automated regression testing of major software systems frequently results in the generation of large volumes of text output files. These files, for the most part the same as the output from testing previous versions of the software system, require extensive review to validate the operation of the modified software. This review is tedious, requiring a lot of attention by highly trained and usually overburdened technical staff members.

The review task can be reduced through the use of standard *Diff tools* that point out the differences between two versions of test output. However, the report of the differences can and regularly does exceed

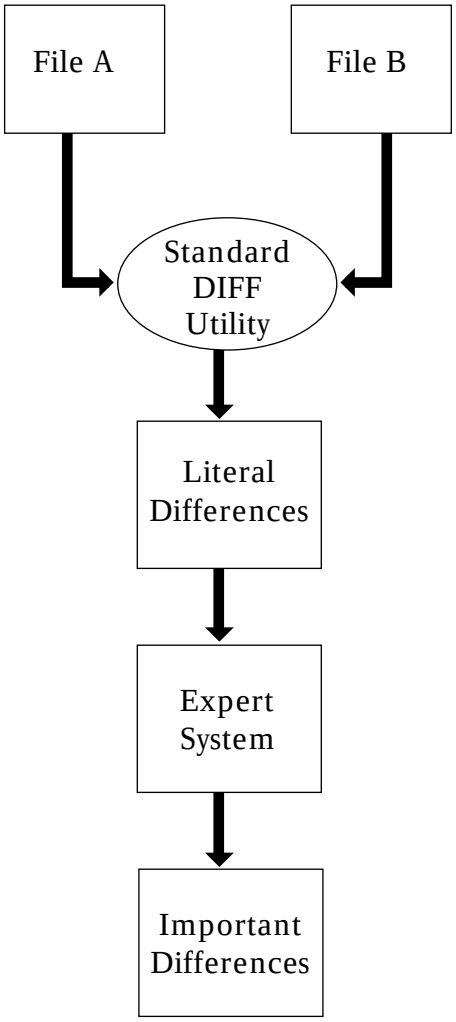

Figure 1. ICT Operation.

the size of the original test output files. Minor differences such as date and time and other information extraneous to the test can overwhelm the Diff process and detrimentally obscure the real and important differences. Other differences, for example, numeric differences, might be within some acceptable range and can be ignored.

Problem Statement

Develop a tool that identifies all important differences and minimizes unimportant differences between the test output from a software system and the expected output from this system.

ICT should significantly reduce the expense, time, and personnel impact associated with the review of regression test output. At the same time, the quality of the review process should be improved. The tool can be used in any situation that produces periodic versions of results that are intended to be nearly the same.

Operation

ICT takes two text files as input and produces output that consists of meaningful differences between the two input files. In addition, log and error files are produced. ICT works in two phases: the literal comparison phase and the intelligent difference elimination phase (figure 1).

Literal Comparison

In the first phase, the two input text files are literally compared. This process is done with a standard Diff tool.

Intelligent Difference Elimination

In the second phase, the literal difference files are processed by an expert system. During the processing of a difference file (containing one or more literal differences), as much text as possible is removed from both parts of a particular difference. (A *difference* consists of two portions, each listing text from one of the two input files.) This text reduction is directed by the rules. It stops either when both portions of a difference have been made identical (that is, the difference was completely irrelevant) or when no further reductions can be achieved (that is, an important difference [according to the rules] has been found).

The output file with the remaining differences is written in the same format as the input so that it can repeatedly be processed with different sets of rules, perhaps with more stringent criteria or more sophisticated rules.

Technical Overview

ICT is a hybrid stand-alone tool with embedded knowledge-based and procedural components. It achieves its intelligence with two major ingredients: string pattern matching and expert system technology.

Text String Pattern Matching

In text string pattern matching, *generalized text strings*, for example, a pattern that matches all dates, are specified using extended regular expressions. *Regular expressions* are a standard and powerful way of defining text string patterns. An excellent description of the regular expression syntax employed in ICT can be found in Aho, Kernighan, and Weinberger (1988).

Expert System

The second ingredient in ICT is a production-rule expert system that captures the test engineer's knowledge and expertise. The rules specify text string patterns that are to be matched on the input text. The rule premises can combine any number of conditions, for example, whether simultaneous matching of several text string patterns occurred (context), whether a particular pattern did not match (exceptions), or what the matched string actually contained. The rules can, for example, dictate to ignore differences in dates or file version numbers only in certain contexts. They can also direct to ignore differences between numbers if the difference is below a certain percentage.

Stepwise Differencing

Another ingredient of ICT is an improvement in the conventional differencing utility. It is a stepwise differencing tool. This tool looks for suitable test step markers and titles that are inserted into the original text files (usually automatically during capture of the output). These markers break the text into sections or steps. When the text files are literally compared, only corresponding test steps that appear in both files are compared with each other. In this way, test steps can be in a different order in the two files or even absent in one of them. The table created by this process is useful in managing the large amount of test information and can become part of the quality assurance document set.

Implementation

In this first implementation, the literal stepwise differencing was performed on a Vax. The expert system was run on a personal computer.

Fortran, DCL on the VAX

The stepwise differencing tool is written in Fortran and DCL and runs on the VAX under VAX/VMS. It uses the VAX difference utility for the literal comparisons.

Personal Consultant on the IBM/PC/AT

The expert system shell chosen for ICT is the Texas Instruments (1987) PC+. It runs on the IBM/PC/AT under DOS. PC+ is written in Scheme Lisp and allows for calls of Lisp routines, which, in turn, can call routines written in procedural languages such as C or Pascal.

The expert system is run from DOS in batch mode. A batch command file invokes the expert system shell as often as desired, each time using a different input file and specifying the knowledge base to be used. During such a session, a single input file containing one or more sets of literal text differences is processed by the expert system.

The rules typically attempt to match a text pattern on the two parts of the difference, removing or blanking out text depending on various conditions. As the last premise of most rules, the now altered parts of the difference are compared. If they have been made identical, the rule succeeds (fires), indicating that this particular difference has been eliminated. If the difference cannot be reduced to two identical parts, a catchall rule with the lowest priority fires, thus satisfying the subframe's goal yet indicating that an important difference was found, which is written to the output file.

Generation of Rules

Using a listing of the literal differences, a test engineer establishes a list of the most common unimportant differences. The tool closely mimics the language of the engineer. A typical statement such as "this difference is unimportant because both output differ only in the date . . . " naturally translates into a rule.

For example, a useful rule compares numbers that are found to be different but might be within an acceptable range for the testing purpose. The rule uses a regular expression to search for floating-point numbers, one from each part of the difference, and computes the discrepancy. Varying percentage differences are allowed depending on the magnitude of the original numbers. If the difference is small enough, the two input numbers are replaced by blanks, and the search continues for the next pair of floating-point numbers. In general, it turned out that writing several simpler rules, each tailored to a similar yet slightly different situation, improved execution time over using just one rather complex rule that dealt with all these situations at once.

Lisp Routines

The rules of the expert system directly call Lisp routines. These Lisp routines perform the bulk of the logistic functions, such as reading and writing files. The use of recursion, lists, and mixed data types provides a convenient and powerful way of specifying regular expressions, which could otherwise easily result in long, cumbersome text strings. In ICT, regular expressions can be represented by text strings, variables representing a regular expression, or (mixed and nested) lists of both. In this way, even long regular expressions can be specified by a list of small, meaningful subexpressions.

C Routines

The Lisp code calls C routines written in Microsoft C. Both Lisp and C perform input-output on the same files but also communicate through variables. All the text pattern matching is performed by C code. It allows for matching a regular expression on a literal difference text to determine whether a match occurred, to retrieve the actual matched string, to replace it with blanks, or to delete it. In addition, the match can be restricted to a substring of the literal difference text. This substring is also specified by a regular expression. C routines that implement text string pattern matching with extended regular expressions were developed at the University of Toronto (Spencer 1986). ICT uses this software with a modification to improve error handling.

Matching is not restricted to single lines as it is in many commonly used text-searching or pattern-matching tools. Rather, the whole text of a literal difference can be matched at once. One can, of course, restrict the match to one line at a time with the proper use of regular expressions.

Frame Structure and Rules

PC+ is organized into frames. A *frame* is a template that contains a collection of goals, premises, rules, facts, and so on. During each session, exactly one instance of the root frame is instantiated. The root frame for ICT is as follows.

```
Root Frame:
==========
Goals:        DONE
Offspring:    DIFFSET
Rules:        (RULE001)
```

Shown here are some of the characteristics of the root frame. Its instantiation puts the goal DONE on the agenda. Through backward

chaining, RULE001 is considered:

RULE001:
========
IF
    (INIT&TRUE)
THEN
    CONSIDERFRAME DIFFSET AND DONE

INIT&TRUE is a Lisp routine. It initializes various variables and helps with the overall system logistics. It always returns true, so that the premise of RULE001 always succeeds. The conclusion causes the frame DIFFSET to be considered (instantiated), and when all instantiations of frame type DIFFSET have been processed, the goal DONE is to be set to true, which ends the session.

All real work is performed by the DIFFSET subframe. In PC+, plus frames can be designed to be instantiated as often as possible, that is, as often as the premise allows.

Frame DIFFSET:
==============
Goal: SET-STATUS
Premise: (LOADNEXT COUNT)
Rules: (RULE002 RULE003 . . .)

When PC+ determines whether this frame should be instantiated for the first (or a repeated) time, the Lisp function LOADNEXT (with a bookkeeping argument) is executed. This function helps with initialization and logistic tasks but, most importantly, loads the next set of literal differences from the input file. Each part of the difference is written to a DOS virtual disk file.

If, however, no more sets of literal differences can be found, LOADNEXT returns false, so that no further DIFFSET frame is instantiated, and the session comes to a conclusion. Otherwise, a new instance of this frame is created. Therefore, a new (and distinct from all previous) goal SET-STATUS is put on the agenda. The backward-chaining rules (RULE002, RULE003, . . .) now come into play. They determine the goal SET-STATUS and, in the process, perform the desired effect of discarding, reducing, or keeping the current set of literal differences.

Here are some rule examples:

RULE002:
========
IF
    (MATCH 1 RE-DATE)
AND
    (MATCH 2 RE-DATE)
AND

```
        (REMOVE&TRUE 12 RE-DATE)
AND
        (COMPARE?)
THEN
        SET-STATUS = "The subsets are equivalent"
```

The first premise calls the Lisp routine MATCH with two arguments, 1 and RE-DATE. RE-DATE is simply a variable representing a regular expression denoting dates, for example:

```
[0-2]?[0-9]-[A-Z][A-Z][A-Z]-19[0-9][0-9] .
```

This regular expression matches dates of the form 8-AUG-1990 or 01-JAN-1991. The other argument, 1, indicates that the regular expression is to be matched on the first of the two parts of the literal difference.

The function MATCH returns true if the match succeeded; otherwise, it returns false. Therefore, the first two premises of rule RULE002 succeed only if both parts of the literal difference contained a date.

The third premise consists of the Lisp routine REMOVE&TRUE. It is only executed when the first two premises succeed. This routine removes (blanks out) the text matched by the regular expression represented by the second argument. This removal is applied to both subsets, as indicated by the first argument, 12.

The fourth premise is a Lisp routine that compares the (now altered) parts of the literal difference and returns true if the two are identical; otherwise, it returns false. *Case sensitivity*, the handling of multiple white spaces, and tabs are controlled by flags that are read from an ASCII file.

If the rule fails at this point, the dates have already been removed, even though another difference remains. However, if the COMPARE? premise succeeds, this set of literal differences does not represent an important difference and is totally eliminated. The current DIFFSET frame's goal (SET-STATUS) is established, the frame processing is concluded, and PC+ goes on to the next set of literal differences.

RULE002 can be paraphrased as "remove dates (of a certain form) from both parts of any literal difference but only if both parts contain a date of that form."

The next rule example performs this task: "Remove times from both parts of the literal difference if both contain times of a given form in a certain context and if the times are within 2 minutes of each other":

```
RULE003:
========
IF
        (MATCH_INRANGE? 1 RE-TIME (LIST "Start: "
                RE_TIME " on node VENUS"))
AND
```

```
        (MATCH_INRANGE? 2 RE-TIME (LIST "Start: "
                RE_TIME " on node VENUS"))
AND
        (DIFFTIME) < 2
AND
        (REMOVE&TRUE 12 (LIST "Start: " RE_TIME " on
                node VENUS"))
AND
        (COMPARE?)
THEN
        SET-STATUS = "The subsets are equivalent"
```

Although essentially similar to the previous rule example, this rule exemplifies the power and conciseness of the ICT rules. The context for the time (of a form specified by a regular expression stored in variable RE-TIME) is itself a rather long regular expression. It is given as a LIST argument, which will be turned into a string by Lisp code through substitution and string concatenation.

The Lisp function DIFFTIME accesses the actual matched strings of the last two matches. From these strings (which represent times), a time difference in minutes is calculated and returned.

The last rule example is the catchall rule of frame DIFFSET. It is necessary if none of the rules could make the two parts of the literal difference identical to each other. In this case, the following rule brings the current frame's processing to a close:

```
RULE004:
========
IF
        SET-STATUS IS NOTKNOWN
THEN
        SET-STATUS = "The subsets have an important difference"
AND
        (WRITE-DIFFSET-TO-FILE)
PRIORITY: -90
```

NOTKNOWN is a PC+ keyword that applies to knowledge facts that have not been established. The low priority of this rule assures that it is the last rule to be tried. As part of its conclusion, this rule writes the possibly reduced remaining parts of the literal difference to the output file.

In summary, each set of literal differences in the input causes a new instantiation of the subframe DIFFSET. The resulting new goal brings rules into play that reduce the parts of the literal differences according to reasoning with matches of text string patterns. Should the two parts of the current literal difference become identical because of this re-

ported reduction, it is immediately discarded, and ICT goes on to the next literal difference. If the two parts of a literal difference cannot be made identical after all rules have been brought to bear on it, then the remaining parts are considered to contain an important difference and are written to the output file.

Thus, the common problem to combine the declarative and procedural worlds is achieved by creating a procedural framework in the declarative expert system shell that allows for the sequential processing of each set of differences and the application of all appropriate rules. Now that the framework is established, it is easy and fast to add new knowledge and expertise.

The current shallow reasoning achieved remarkable results in the first use of ICT. However, knowledge can also be structured in subframes and rules that achieve high degrees of reasoning complexity as appropriate for the software-validation task at hand.

Experience

ICT was used in the spring of 1989 to aid in the software-validation process for a customer. The software to be tested consisted of a large real-time application of about 50,000 lines of Fortran code used to monitor the fuel performance of a nuclear reactor. Proper validation of this system was an important part of the development cycle.

ICT system was applied to the large volumes of output generated by an integration test run. Specifically, the output of a current test run had to be compared to a benchmark case. The size of each of these files was typically around 2.5 megabytes (MB).

The literal comparison produced files whose combined size was also around 2.5 MB. These files were substantially reduced by the ICT system. A surprisingly small number of rules resulted in a dramatic reduction of output. With only 28 rules, the literal differences were reduced to 18 percent of their original size (counting all nonblank characters).

Included in this reduction were many differences that typically started with about 100 floating-point numbers in each set and that ended mostly blank, with a few out-of-range numbers easily visible. Compare figure 2 with figure 3.

By eliminating most of the differences that are not important to a test engineer, ICT was able to focus the engineer's attention on the essential differences between the test run and the benchmark case. ICT achieved large savings in time and replaced spot checking of the past with exhaustive checks.

In addition, the fact that ICT divides the large, unwieldy test file into

```
File ABC_01.DAT: lines 1733 - 1739
0.727 0.685    0.668    -1.36
0.726 0.683    0.665    -1.37
0.720 0.682    0.651    -1.38
0.719 0.664    0.648    -1.39
0.717 0.662    0.646    -1.46
0.715 0.660    0.644    -1.41
*****
File ABC_02.DAT: lines 1842 - 1847
0.727 0.686    0.668    -1.36
0.726 0.683    0.669    -1.37
0.729 0.682    0.651    -1.39
0.719 0.664    0.648    -1.39
0.717 0.662    0.646    -1.46
0.715 0.660    0.643    -1.46
```

Figure 2. Literal Differences.

File ABC_01.DAT: lines 1733 - 1739

-1.41

*****

File ABC_02.DAT: lines 1842 - 1847

1.46

Figure 3. Important Differences.

small, logical parts (test steps) and generates a table of contents for these test steps was helpful in managing the amount of data involved.

During the original experience with the ICT system, all work was performed by the ICT development team. As a result, it is believed that greater efficiency in operation was experienced than had less highly trained personnel been used. It is expected that the next application of the system will use personnel who are not aware of the details of the

operation of the system, thus giving a more realistic picture of the savings that might result from long-term operation of ICT.

Payoffs

Customer savings were established by the test engineer who has reviewed the tested system for the past three years. He estimated that ICT reduced the time required for a complete review from 200 to 4 resource hours. Operation of ICT adds 2 resource hours.

Although the time savings were important to the customer, more important was the improved quality of the review process. Most important to the test engineer was that spot checking and error-prone reviews were replaced by reviews that are exhaustive, repeatable, and less reliant on expert personnel. Also much appreciated was the reduction in tedium.

For minor software revisions, in the past, only the portions of the test output that were thought to be relevant were reviewed, usually requiring about 20 resource hours. Now a complete review is performed each time.

The use of ICT revealed several software problems that had remained undetected in previous regression tests. Resolution of these problems at this point in the development cycle certainly reduced overall system costs. The customer response has been enthusiastic.

Development Schedule and Costs

The development time for a prototype was about four resource months (two part-time persons for several months). It included the procedural framework (Lisp code, C code, Lisp-C interface) and a few rules.

The necessary hardware and software consisted of a PC/AT with a hard disk and extended memory, PC+, Microsoft C, and some utility programs. The total expenditure for these items was $4200.

The transfer to actual field use was done in three stages. First, the rules specific to the software-validation task at hand were developed together with some enhancements of the ICT framework. This system processed output from a Prime computer's Diff utility but did not yet use the stepwise differencing. In the second stage, the system was adapted to process output from a VAX Diff utility. Third, the stepwise differencing capability was added. During these three stages, two software revisions were processed by ICT. The total time effort was five resource months.

The established framework allows for easy and fast adaption to the validation of different software systems; only the knowledge base rules have

to be replaced. Because a small number of rules achieved remarkable results, it is estimated that this process would take only a few weeks.

As ongoing work for the customer, it is planned to port ICT to a different platform, refine the current knowledge base framework, and create several new rule bases for validation of other software systems. The new platform will be Nexpert Object from Neuron Data running on a VAX computer under VAX/VMS.

Conclusions

The following conclusions were derived from our experience in developing and using ICT: First, expert system technology merged with text pattern matching is successful in partially automating the review of regression test output. Second, the tool increases the quality of the review process. The review focuses on the relevant test differences. Spot checking can be replaced by exhaustive review. The acceptance criteria for the automated review are unambiguously documented. The review process is more accurate and repeatable. Third, ICT minimized review tedium and maximized review engineer effectiveness. The morale of the review engineer was greatly improved. Fourth, the established framework allows for the quick and straightforward creation of knowledge bases for use in new software validation tasks. The knowledge base creation process is well adapted to the way test engineers express their expertise. The overall experience with ICT has been extremely positive, as attested to by an enthusiastic customer response.

Acknowledgments

This project would not have been started or successfully completed without the wise insights of Harold Brown, who is currently at the Hewlett Packard Laboratory, Palo Alto, California.

Portions of this work were funded by GE Consulting Services, San Jose, California, which also provided a nurturing environment in which to pursue the effort.

References

Aho, A. V.; Kernighan, B. W.; and Weinberger, P. J. 1988. *The AWK Programming Language*. Reading, Mass.: Addison-Wesley.

Spencer, H. 1986. Extended Regular Expression Software, University of Toronto.

Texas Instruments. 1987. Personal Consultant Plus (PC+), version 3.0, and PC Scheme, version 3.0.