AUI - the Debugger and Assertion Checker for the Awk Programming Language[1]

Mikhail Auguston, Subhankar Banerjee, Manish Mamnani, Ghulam Nabi, Juris Reinfelds, Ugis Sarkans, Ivan Strnad

Department of Computer Science
New Mexico State University
Las Cruces, NM 88003, USA

Abstract

This paper describes the design of Awk User Interface (AUI). AUI is a graphical programming environment for editing, running, testing and debugging of Awk programs. The AUI environment supports tracing of Awk programs, setting breakpoints, and inspection of variable values. An assertion language to describe relationship between input and output of Awk program is provided. Assertions can be checked after the program run, and if violated, informative and readable messages can be generated. The assertions and debugging rules for the Awk program are written in a separate text file. Assertions are useful not only for testing and debugging but can be considered as a mean for program formal specification and documentation.

1 Introduction

The execution model of Awk programming language differs from the conventional imperative languages like C. According to [1] the behavior of Awk program can be described as follows. An Awk program is a sequence of patterns and actions that describe what to check for in the input data and what to do when it is found. A pair of a pattern and a corresponding action is called a *rule*. The input file is considered to be a sequence of records (lines) and each record contains a sequence of fields. Awk program performs a loop that reads input file lines one by one and checks whether the line read matches any of the rule patterns. When a matching line is found, the corresponding action is performed. A pattern can select lines by combination of regular expressions and comparison operations on fields, strings, numbers, and array elements. Actions may perform arbitrary processing on selected records; the action language looks like C but there are no declarations, and strings and numbers are built-in types. Awk scans input file and splits each input line into fields automatically. The current input record can be referred to in the program as $0, and fields of this record - as $1, $2, and so on. The built-in variable NF contains the number of fields in the current record, the NR variable provides the number of the current record. An example of Awk program that prints the largest first field and the line that contains it (assumes some $1 is positive):

```
$1 > max { max = $1; maxline = $0; }
END { print max, maxline, "\n total lines:" , NR }
```

1. This work has been supported by the Department of Defense research grant #0-93-35

The notions of file, record (line), record field, and number and string are the most essential for reasoning about behavior of Awk program.

There are quite a few effective debuggers for languages like C and Pascal, but we have not found any Awk debuggers. A visual programming environment with debugging facility for Awk can therefore be justified.

2 Interface Design

2.1 Software Platform

The motivation for selecting an appropriate software and hardware platform comes from the following factors: multipoint interruption and continuation of the execution of Awk programs. We decided to modify the current `gawk` interpreter designed by Free Software Foundation so that the interpreter may be stopped at any arbitrary point.

The interface is developed in the C language and uses Motif toolkit [4] from the Open Software Foundation (OSF). The Motif toolkit is based on the X Toolkit Intrinsic (Xt) [5], which is the standard mechanism for many of the toolkits written for the X Window System.

A modified version of the `gawk` interpreter is used to provide interruptions in execution at arbitrary points. This interpreter also allows us to examine the internal state of execution at any point. Communication between the interface and the interpreter is achieved through pipes, via a set of protocols.

Initially, when the user wishes to execute an Awk program, the interpreter is forked, and the program and input data are passed to the interpreter. After syntax checking of the program, the interpreter waits in a loop for messages to arrive. When the interpreter receives a signal, it performs a particular action and then notifies the interface about the completion, accompanied by the associated data (all communications are bidirectional).

The interpreter and the interface communicate for various user requested tasks. All the communication is initiated by the interface in response to the user's actions.

2.2 Implementation Problems and Our Solutions

For partial selections of program text and input data, and for setting breakpoints, we needed the capability of selecting multiple non-contiguous lines in the editors. Motif text widget does not have this capability, any try of new selection deselects the previous selected text. We provide a separate graphical area to the left of the editor's window to cater the needs of non-contiguous selection. When the user clicks in the graphical area, the coordinates of that point are stored in an internal data structure for future use. A selection or a breakpoint marker is displayed in the graphical area at the click point. If the text window is scrolled or resized, all the selection marks in the graphical area are redrawn. When needed, the selected program and the input data are passed on to the interpreter. If the user edits the input file or the program, the previously made selections are discarded.

2.3 Awk User Interface Windows

The interface has two modes: *Edit* and *Execute*. The user interface at start-up is by default in the *Edit* mode. This allows the user to write an Awk program or open an existing program into the text editor. The text editor contains basic editing functions such as cut, copy, paste, and clear. Some or all of the program can be selected for execution. The selections can be made by clicking in the graphical area on the left hand side of the program text. The entire program can be selected or deselected with the buttons below the editor window.

The user opens the input file editor, where a data file could be created or edited. After the selection of lines to be executed from the input file and the Awk program, the user can change the interface to the *Execute* mode. If the user does so, the `gawk` interpreter is started and syntax checking on the Awk program is performed. If any syntax errors are found, the user is notified and interface returns to the *Edit* mode.

In the *Execute* mode, both editors are switched to a 'read-only' state. The editors display only the selected lines of the Awk program and of the input file but the remaining lines are still available for future use. They are not discarded, only temporarily hidden from view. The buttons below the program editor are swapped to execution functions. The user can add breakpoints to both the Awk program and the data file again using the graphical area on the left side of the text. Breakpoints can be dynamically added and deleted at any point in the *Execute* mode.

The user may choose to execute the program in a *step* or a *run* mode. In *step* mode, the program is executed linewise on both the program and data. The currently active data line and the program line are highlighted (Figure 1) The output data of the whole execution session is displayed in the

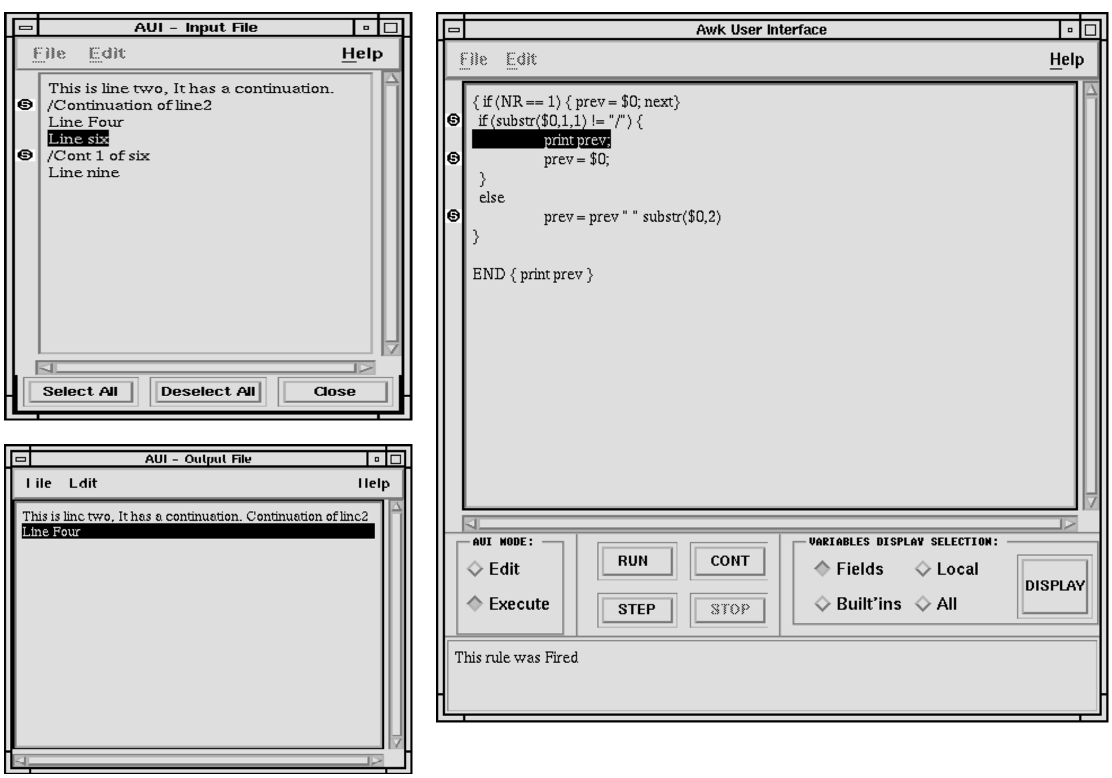

Figure 1: Interface in Execute Mode. Current program segment, input line, and the output are highlighted

output editor window. The output of the currently executing program line is also highlighted. The message area informs the user whether or not the current rule was fired.

In the *run* mode, the program is executed from the beginning until a breakpoint is reached in the program or data file, or until the program terminates. The currently active data line, the currently executing program line and the resulting output are highlighted in the run mode as well.

Execution can be continued from a breakpoint. Execution can also be stopped dynamically at any arbitrary point by using the STOP button. At a stop point the user can examine the values of variables. The variables are displayed in the Variable Display Window. The Variable Display Window is updated at every stop point

All errors are displayed in dialog boxes. If there are any syntax errors in the program, they are displayed in the error dialog and the interface remains in the *Edit mode* for the user to correct them. Run-time errors are reported in the error dialog box and execution is stopped at that point. The user can examine the values of any variables at that point and then go to the *Edit mode* or rerun the program from start.

3 Assertion Language for Awk

The properties of the Awk program can be checked by considering relationship between the input and output files of the program. The assertion language is provided to be the mean for precise specification of Awk program input/output relationship. The assertion can be checked automatically after the program is run. Assertions can be used as conditions for debugging rules that can provide informative messages helpful for the Awk program debugging.

The main notions of the assertion language follow those of the Awk language. The notions of *file*, *record*, and record *field* are the basics.

The semantic model of Awk assumes that the file record is a pair (file, record number within file). This makes it possible,e.g. to define a function like LAST_PRECEDING(`r`) that given a record `r` yields another record preceding `r` in the file.

Typically the assertion describes some condition that holds for records and fields of input and output files. Operations over fields comprise =, !=, >, <, >=, <=. Fields can be compared with string constants and other fields. Strings are represented as it is required in the Awk and C, e.g. "abc".

The SUBSTR built-in function provides the facility to get arbitrary substring of a field or a record for further operations. Arithmetic operations +, -, *, / , and % can be performed on record fields.

Quantifiers can be applied over records within file and over numbers within number interval. Aggregate operations are useful for selecting a sequence of records from a file or another sequence.

For instance, the following aggregate selects from the file `data1` all records where the first field is different from `'\"'`.

```
[ x: RECORD & x . $1 != '\"' FROM FILE data1 ]
```

Aggregate can be considered as a loop over its base which in turn may be an aggregate. The variable behaves as a loop variable when traversing the records of the base. If a boolean expression is given after the & symbol the resulting aggregate will contain only those records from the base for which the boolean expression evaluates to TRUE. Quantifiers over records can be applied on such sequences as well.

An assertion can be supplied with the SAY and ONFAIL SAY statements that can output informative messages when the assertion is true or false, correspondingly. The assertion language expressions can be used in the SAY statements in order to provide useful information about input/output file values. Such set of assertion and SAY statements is called a *rule*.

Assertions and rules can be used when debugging the Awk program, for regression testing and as a useful mean for program maintenance.

The assertions and debugging rules for the Awk program are written in a separate text file. This is convenient when maintaining Awk program. There may be several assertion files for the same Awk program. Assertions are useful not only for testing and debugging but can be considered as a mean for program formal specification and documentation. They are useful for regression testing and for program maintenance as well.

4 An Example of Using the Assertion Language for Awk.

The input file contains a list of all states of U.S.A. There are 50 records separated by newlines, one for each of the states. The number of fields in a record is variable. The first field is the name of the state, and the subsequent fields are names of neighbor states. Fields are separated by tabs. For example, the first two records in the database are

```
Alabama Mississippi Tennessee Georgia Florida
Alaska
```

The task is to color the U.S.A. map in such a way that any two neighboring states are in different colors. We will do it in a greedy manner (without backtracking), assigning to every state the first possible color. The Awk program for this task is the following:

```
# Greedy map coloring
BEGIN { FS= "\t"; OFS= "\t" # fields separated by tabs
     color[0]= "yellow"  # color names
     color[1]= "blue"
     color[2]= "red"
     color[3]= "green"
     color[4]= "black"
     }

{    i=0
     while (a[$1,i] ) i++      # find first acceptable color for
                               # state $1
     print $1 "\t" color[i]    # assign that color
     for (j=2; j<=NF; j++) a[$j,i]=1# make that color
                                    # unacceptable for
                                    # states $2..$NF
}
```

We can check the correctness of the coloring using the following assertion:

```
/* Checks the correctness of map coloring - any two neighbor
```

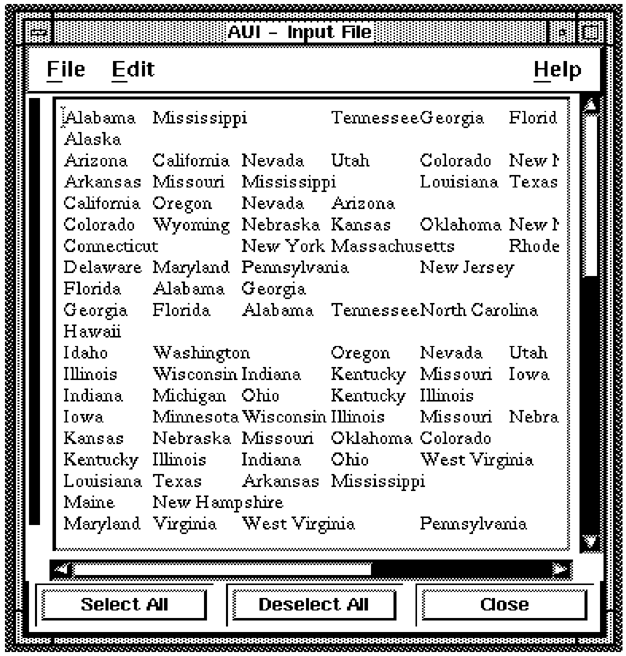

Figure 2. AUI input window.

```
 states should be colored in different colors */
FOREACH r1: RECORD FROM FILE input
     (EXISTS r2: RECORD FROM FILE output
          (r1.$1 == r2.$1 AND
           FOREACH i IN 2..FIELD_NUM(r1)
               (EXISTS r3: RECORD FROM FILE output
                    (r3.$1 == r1.$i AND r3.$2!=r2.$2)
               )
          )
     )
SAY "Map colored correctly"
ONFAIL    SAY r1.$1 "and" r1.$i "are of the same color"
          SAY "although they are neighboring states"
```

The evaluation of the boolean expression under the quantifier is performed in a "lazy" mode, i.e. until the TRUE value is obtained for EXISTS quantifier, or FALSE for FOREACH quantifier. The value of variable bound by the quantifier is retained and can be used in SAY clause attached to the corresponding assertion. This is a convenient way to get an informative message if the assertion fails (we can print the value of variable for which e.g. FOREACH condition fails the first time).

For every state from input data (record `r1`) we find the corresponding record on output (record `r2`). Then we find all records in output corresponding to neighbors of `r1.$1` (record `r3`) and check if the colors are different (`r2.$2 != r3.$2`). If the assertion is violated, we can use the last values of `r1` and `i` and determine which neighboring states have the same color.

The Awk program above works correctly (Figure 3).

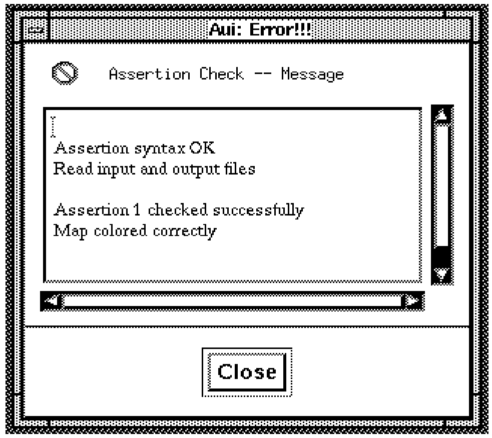

Figure 3. Assertion checker output - success.

To test ONFAIL part of the assertion we change the while loop to

```
while (a[$1,i] && i<3) i++
```

That means we do not allow more than 4 colors. If the database is sorted alphabetically, the greedy algorithm needs 5 colors and we get an error - Missouri and Oklahoma get the same green color. We can detect it using the assertion (Figure 4).

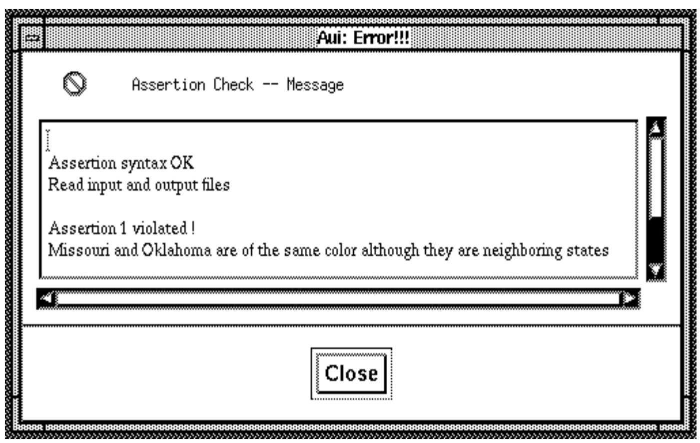

Figure 4. Assertion checker output - failure.

5 Data Verification

The assertion language can be used for input data verification as well. The following assertion checks the integrity of the input file. For example, if there is a record

```
New Mexico ... Arizona ...,
```

there also has to be record

```
Arizona ... New Mexico ...
```

on the input file.

```
FOREACH r1: RECORD FROM FILE input
```

```
      (FOREACH i IN 2..FIELD_NUM(r1)
            (EXISTS r2: RECORD FROM FILE input
                  (r1.$i==r2.$1 AND
                   EXISTS j IN 2..FIELD_NUM(r2) (r1.$1 == r2.$j )
                  )
            )
      )
SAY "Correct data"
ONFAIL SAY r1.$1 "and" r1.$i " - one-sided relation" r1
```

We traverse for each record (base variable `r1`) all neighbors (fields `$2..$NF`) and check the existence of corresponding record for the neighbor (record `r2`) and existence in that record the name of the state currently under consideration.

If we delete from the New Mexico record, e.g., Utah, but leave New Mexico in the record for Utah, we'll get the appropriate message from the assertion checker.

6 Another Examples of Awk Program and Debugging Rules

Input file Data1; lines with dittos in column one.

ABCD 12345 LINE1
AA XXX RRR SSS
" YYY JJJ KK
" LINE4
GDE AA BB CC
" DDDD EEEE
WXYZ
MT AAA BBB
" JJJ KKK
JK

The Awk program has to Replace dittos at beginning of line with data above it:

```
# cmd1: awk script to replace dittos
# usage: awk -f cmd1 data1 > out1
$1 != "\"" { tmp = $1; print $0; next}
          # hold field 1 for possible use
{
ind = index($0, $2)# starting position of field 2
print tmp, substr($0, ind)
}
```

Output file out1; output of above

ABCD 12345 LINE1
AA XXX RRR SSS
AA YYY JJJ KK

AA LINE4
GDE AA BB CC
GDE DDDD EEEE
WXYZ
MT AAA BBB
MT JJJ KKK
JK

Assertion (Rule) 1.

```
LEN( FILE data1) == LEN( FILE out1)
   SAY "The number of records in each of files is:"
      LEN( FILE data1);
/* The number of records is the same in data1 and out1 files */
```

The LEN function yields the number of records for file or record aggregate, the string length for a record or string. When the assertion checker is executed the following message is output.

```
Assertion 1 checked successfully
The number of records in each of files is: 10
```

Assertion (Rule) 2.

```
FOREACH a: RECORD FROM FILE data2
   ( EXISTS b: RECORD FROM FILE out2
      ( REC_#(a) == REC_#(b)  AND
        FIELD_NUM( a) == FIELD_NUM( b)  AND
         FOREACH f IN 2 .. FIELD_NUM(a)
           (a. $f == b. $f )))
ONFAIL SAY "For input record" a
          "there does not exist corresponding output record"  ;
```

REC_# returns the record number within file, e.g. for the first record in the file it yields 1, for the last record within file the REC_# is equal to the LEN(file).

Let the data2 file be as follows:

ABCD 12345 LINE1
AA XXX RRR SSS
" YYY JJJ KK

and out2 file be (e.g. because of a bug in the Awk program):

ABCD 12345 LINE1
AA XXX RRR SSS
AA JJJ KK

The field YYY is missing in the last record. In this case the assertion checker for the Assertion2 outputs a message like:

```
Assertion 2 violated !
For input record
'\" YYY JJJ KK"  (File: data2 , Record # 3 )
there does not exist corresponding output record
```

Assertion (Rule) 3.

```
FOREACH b: RECORD FROM FILE out1
(
   (EXIST a: RECORD FROM FILE data1
      (  ( a.$1 != '\"'   AND  b. $1 == a. $1
            AND REC_#(a) == REC_#(b) ) )
   OR
   ( b. $1 == LAST_PRECEDING c: RECORD & c. $1 != '\"'
            (FILE data1.REC_#(b)). $1  )
)
ONFAIL SAY "For output record" b
        SAY "The first field does not have the proper value"  ;
```

For the LAST_PRECEDING function the `FILE data1.REC_#(b)` represents the base record with respect to which the last preceding record `c` in the file is looked up, such that satisfies the condition `c. $1 != '\"'`. If such preceding record does not exist within file the value of the LAST_PRECEDING function is an empty record.

Rule 4.

This Rule is used in order to print the content of the files.

```
TRUE
   SAY "The input file data1 is:"   SAY   FILE data1
   SAY "The output file out1 is:"   SAY   FILE out1    ;
```

the condition of this Rule is simply TRUE, i.e. the SAY clause is executed unconditionally. The result of running the assertion checker on this Rule may be e.g.:

```
Assertion 4 is checked successfully

The input file data1 is:
'ABCD 12345 LINE1' (File: data1 , Record # 1)
'AA XXX RRR SSS' (File: data1 , Record # 2)
'\" YYY JJJ KK' (File: data1 , Record # 3)
The output file out1 is:
'ABCD 12345 LINE1' (File: out1 , Record # 1)
```

```
'AA XXX RRR SSS' (File: out1 , Record # 2)
'AA YYY JJJ KK' (File: out1 , Record # 3)
```

Rule 5

This Rule demonstrates the usage of aggregate operation.

```
TRUE
SAY "The records of the data1 file with not empty first field:"
SAY [ x: RECORD & x . $1 != '\"' FROM FILE data1 ] ;
```

The result of running assertion checker on this Rule is:

```
Assertion 4 is checked successfully

The records of the data1 file with not empty first field:
'ABCD 12345 LINE1' (File: data1 , Record # 1)
'AA XXX RRR SSS' (File: data1 , Record # 2)
'GDE AA BB CC' (File: data1 , Record # 5)
'WXYZ' (File: data1 , Record # 7)
'MT AAA BBB' (File: data1 , Record # 8)
'JK' (File: data1 , Record # 10)
```

7 Conclusions

We have developed a graphical user interface for the Awk programming language that provides a user friendly environment over the current means of developing and running an Awk program.

The assertion language for Awk is a powerful tool for checking various properties of input/output relations. It can also be used to verify the correctness of input data. In principle the assertion language could be used as a stand-alone tool to check relations between several files as well.

References

[1] Aho, A. V.; Kernighan, B. W.; and Weinberger, P. J., The AWK Programming Language, Addison-Wesley Publishing Company, 1988.

[2] CenterLine Software, CodeCenter User's Guide, CenterLine Software, Inc., 1986.

[3] SUN Microsystems, SunOS Reference Manual, Sun Microsystems, Inc.,1990.

[4] Nye, A.; and O'Reilly, T., X Toolkit Intrinsics Programming Manual, OSF/Motif Edition, O'Reilly & Associates, Inc., 1990.

[5] Heller, D.; Ferguson, P. M.; and Brennan, D., Motif Programming Manual, O'Reilly & Associates, Inc., 1994.