DEVELOPING AN AUTOMATED PROGRAM CHECKER

STUDENT PAPER

Julia Isong
Department of Computer and Information Sciences
Florida A&M University
Tallahassee, FL 32307
jisong@cis.famu.edu

ABSTRACT

This paper demonstrates the use of an automated program checker as a fast and effective technique for checking numerous programming assignments submitted by students. The process of designing and developing the checker is illustrated by a sample assignment. The paper concludes with a set of guidelines for preparing an assignment specification that is amenable to automated checking.

1 INTRODUCTION

The purpose of this project is to develop a fast and efficient way to grade a large number of student programs. One way of doing so is to use an automated checker [1,2,3]. The challenge is to develop a testable specification, so that the specification provides an adequate basis for designing an effective checking process. This paper shows a process for generating a Unix-based program checker that assesses the functional correctness of the program. The approach is similar, though less sophisticated than the TRY system [3]. The checker does not assess other aspects of the assignment solution--efficiency, style, complexity and test data adequacy—as performed by the ASSYST system [1].

Automated checking has the advantages of being consistent, thorough, and efficient. Each program will be checked with the same level of efficiency and the resulting assignment grade will be based on the same standards. In addition, the checker produces a detailed report of deductions for each student's program; this report is usually enough to answer most questions such as "What did I do wrong?" An additional benefit to students is an appreciation for issues of software engineering such as software specification, software standards, and software testing [2]. The following steps are required to construct the automated program checker.

1. Create an assignment specification
2. Develop test cases from the assignment specification.
3. Develop a grading plan based on the assignment specification and test cases.
4. Design the program checker based on the grading plan and test cases.

A single example will be used throughout this paper. The remaining sections are organized as follows. Section 2 introduces an assignment specification that will be used throughout the paper. Section 3 presents an approach to generating test cases based on the assignment specification. Section 4 shows how to develop the grading plan specifying the correctness checks and deduction schedule that will be applied. Section 5 illustrates the design of the automated program checker that implements the grading plan. Section 6 presents some guidelines for developing a specification amenable to the developing an automated checker. Section 7 contains conclusions and identifies on-going and future work.

2 EVOLVING THE ASSIGNMENT SPECIFICATION

The purpose of the assignment specification is to tell students exactly what is expected of them. The initial specification will not necessarily be the specification that the students will see, since a series of refinements may be necessary to ensure that the assignment can be checked using an automated checker.

Table 1 contains the final specification for an assignment to compute employee pay based on hours worked, employee classification (hourly or salaried), and pay rate. Note that the specification includes explicit requirements for output formatting, ranges of valid inputs, and business rules for performing calculations.

Table 1: Program Specification – Pay

OBJECTIVES: Compute pay based on hours worked and pay rate.

PROGRAM REQUIREMENTS

1. The program prompts for hours worked and pay rate, in the format:

```
Enter Hours Worked:
Enter Pay Rate:
```

2. Pay rate, pay, and hours must be decimal numbers.
3. Hours must be between 0 – 112, or return –1.
4. Pay rate must be between 5.15 – 50.00, or return –1.
5. Hourly employees earn $5.15 - $20.00 per hour.
6. Salaried employees earn $20.01 - $ 50.00 per hour.
7. For hourly employees, overtime hours are those over 40 hours.
8. For salaried employees, overtime hours are those over 50 hours.
9. The overtime pay rate is 1.5 times the regular pay rate.
10. Program output must be in the format:

```
Hours = xxx.x, Rate = xx.xx, Pay = xxxx.xx
```

3 CREATING TEST CASES

Checking a program for functional correctness is a problem in testing. The first step in the process is to design test cases to be used to demonstrate correctness. The assignment specification is converted to a decision table, as shown in Figure 2. The columns of the decision table identify possible input combinations (in the top portion), and the way the program must respond (in the lower portion). This information is the basis for the test cases in Figure 3.

Figure 1. Decision Table for Pay

Hours >= 0	Y	Y	Y	Y	N	-	-	-
Hours <= 112	Y	Y	Y	Y	-	N	-	-
Rate >= 5.15	Y	Y	Y	Y	-	-	N	-
Rate <= 50.00	Y	Y	Y	Y	-	-	-	N
Hours > 50	Y	Y	N	N	-	-	-	-
Hours > 40	Y	Y	N	-	-	-	-	-
Rate <= 20.00	Y	N	Y	N	-	-	-	-
Pay = Hours * Rate	-	-	X	X	-	-	-	-
Pay = Rate * 40 + 1.5*Rate*(Hours-40)	X	-	-	-	-	-	-	-
Pay = Rate * 50 + 1.5*Rate*(Hours-50)	-	X	-	-	-	-	-	-
Pay = -1					X	X	X	X
Function Behaviors (equivalence)	1	2	3	4	5	6	7	8

The test case design process requires that one test case be created for each column of the decision table. These *nominal* test cases test the distinct behaviors of the program, and represent the minimum amount of testing. An example of a nominal test case is the triple (50 hours, $10 per hour, expected pay $550), a typical set of values. Other techniques for test case generation, such as *boundary testing*, may be considered when more rigorous testing is desired. An example of boundary test cases involving the specification element `Rate <= 20.00` is the inclusion of test cases with rates that lie at and near the boundary of $20.00, namely, $19.99, $20.00 and $20.01. The last four columns in Figure 2 represent *error testing* to determine the behavior of pay when the preconditions in Figure 1 are not met; in these cases, the expected pay is –1.

Figure 2. Test Cases

	1	2	3	4	5	6	7	8
Rate	10.00	30.00	10.00	40.00	10.00	10.00	5.00	51.00
Hours	50	60	32.6	45	-5	113	40	50
Expected Pay	550.00	1950.00	326.00	1800.00	-1	-1	-1	-1

4 DEVELOPING THE GRADING PLAN

The *grading plan* identifies which expected results to check for, and the numeric penalty assessed for failure to produce the expected result. The grading plan need not attempt to thoroughly test the program, but should target significant outputs reflecting

the assignment objectives. Some of the decisions the instructor must make when formulating the grading plan include:

1. How much effort to expend doing the checking
2. Which test cases to include
3. Other aspects to check (e.g., format, spacing, standard outputs, etc…)

Three checking strategies are used. The *presence* strategy simply checks whether a specific value occurs at least once somewhere in the output stream produced when the program executes (an excerpt of the output stream is shown below). According to the assignment specification in Table 1, the output stream should contain at least one message containing the pattern "`Hours =` ". Whether or not there are more is not important, the format is.

```
Enter total number of hours worked:
Enter the pay rate:
Hours = 55.0,  Rate = 10.00,  Pay = 625.00
Enter total number of hours worked:
Enter pay rate:
Hours = -5.0,  Rate = 10.00,  Pay = -1.00
Enter total number of hours worked:
Enter the pay rate:
Hours = 113.0,  Rate = 10.00,  Pay = -1.00
```

A second strategy, *aggregate* checking, focuses on a specific number of occurrences of the targeted result. For example, checking that the preconditions (0 <= Hours <= 112 and 0 <= Rate <= 50.00) are handled correctly can be determined by counting the number of times "`-1`" appears in the output stream. The final strategy, *instance* checking, searches for the results from a specific test case. For example, in the output stream, there should be *exactly one* message "`Hours = 50, Rate = 10.00, Pay = 550.00`". The instance strategy, since it is more detailed, requires greater effort, but is more fine-grained than the aggregate or presence strategies. Figure 3 contains a grading plan for the assignment Pay.

Figure 3. Grading Plan for Assignment Pay

```
(1) Check for format of messages – correct data captions
    Expect                  Deduction
    "HOURS = "                2
    "PAY = "                  2
    "RATE = "                 2 (presence check)
(2) Check for correct calculations for test cases
                        Expected
     HOURS    RATE       Pay        Deduction
      55     10.00     625.00         3
      55     15.00     862.50         3
     32.6    10.00     326.00         3
     48.9    15.00     733.50         3 (instance check)


      -5     10.00    -1              3
     113     10.00    -1              3
      40      5.00    -1              3
      50     25.00    -1              3 (aggregate check)
```

5 DEVELOPING THE PROGRAM CHECKER

The program checker executes the student program using an input stream containing test cases. The output from the student program is written to the program's output stream. The checker performs pattern-matched searching on the program's output stream to detect discrepancies between expected and actual outputs from the student program. The checker writes details about each discrepancy to the student's grading report.

The grading plan and the test cases are the bases for coding the program checker in the C shell script language. Figure 4 contains excerpts from the program checker that implements the grading plan, and shows how presence, aggregate, and instance checking are performed. The central construct in the checker code is the Unix grep command, which searches for patterns and returns a value reflecting success or failure to satisfy the checking strategy. For a presence check, the `grep` command checks the output stream for at least one message containing the pattern "`Hour =` ". If this pattern is not found, a two-point deduction is made.

Figure 4. Program Checker Shell Script for Assignment Pay

```
#-----------------------------------------------------------------------
# File name :  check
# Purpose   :  Check program, deducting points for errors
# Invocation:  check program instream outstream
#-----------------------------------------------------------------------
#-| Start checking with 100 points.

set pts = 100

#-| Presence Check - format "Hours = "
#-| Deduct 2 if not found.
set count=`grep -c "Hours =.* " $3`
if ($c< 1) then
  @ pts = $pts – 2
  echo " "
  echo "BAD FORMAT ('Hours = ') DEDUCTION 2 PTS"
endif
```

```
#-| Aggregate Check - four -1's :
#-| Deduct 3 for each not found
  set c= `grep -c ".*-1.00.*" $3 `
  @ D = 3 * (4 - $count)
  @ pts = $pts - $D
  if ($c!= 4) then
    echo " "
    echo "WRONG NUMBER OF '-1.OO' - EXPECTED 4 - DEDUCTION $D PTS"
  endif


#-| Instance Check - '55 10.00 625.00' :
#-| Deduct 3 if not found.
  Set c= `grep -c "55.*10.00.*625.00" $3 `
  if ($c!= 1) then
    @ pts = $pts – 3
    echo –n "WRONG PAY -- EXPECTED '625.00' for 'Hours = 55, Rate = 10.00'"
    echo " DEDUCTION 3 PTS"
  endif
echo " "
echo "Final Program Score = $pts"
     exit
```

6 GUIDELINES FOR WRITING THE ASSIGNMENT SPECIFICATION

An automated program checker can be constructed in a straightforward manner when the assignment specification is testable. The key to having a testable assignment specification is to make it as explicit as possible. Being explicit means stating the general and giving examples of the rule. The testability of the assignment specification is enhanced by these guidelines.

- Function – State clearly what the program must do. When there may be confusion, include an example to accompany the specification.
- Inputs – Identify each input, its data type and domain (range of acceptable values). Also specify the sequence in which inputs must be presented.
- Outputs – Identify each output, its data type and domain. Specify the format of output values (e.g., field widths), output captions and labels, messages containing multiple values, and other formatted output.
- Error handling – Establish boundaries (e.g., preconditions), and specify how the program must respond when these boundaries are not satisfied. Also include potential run-time errors that may occur.

Finally, one must be prepared to evolve the specification based on the insights gained formulating the grading plan and designing the program checker. Often, simple changes in the specification lead to a simpler and faster checker.

7 SUMMARY AND FUTURE WORK

Using the automated checking testing approach can be beneficial to both instructor and student. The instructor can grade more assignments in a more efficient manner, and every student can be guaranteed the same checking standards. Students who are exposed to this approach will learn to anticipate the way programs are graded and will do a better job designing their solutions. Future work includes automating the generation of the program checker based on checking strategy and elements of the assignment specification. A near term activity is providing assistance to teachers in constructing grading plans for their assignments.

ACKNOWLEDGEMENTS

The author, who is an undergraduate researcher funded under NSF grant EIA-9906590, acknowledges the patient mentoring received from Dr. Edward L. Jones over the past year.

REFERENCES

[1] Jackson, D. and Usher, M. Grading Student Programs Using ASSYST. *Proceedings SIGCSE '97*, (1997) 335-339.

[2] Jones, E.L. Grading Student Programs: A Software Testing Approach. *Journal of Computing in Small Colleges 16,* 2 (January 2001), 185-192.

[3] Reek, K.A., "The TRY System – or – How to Avoid Testing Student Programs," *Proceedings SIGCSE Bulletin vol.21,* 1 (February 1989), 112-116.