AN EXPERIMENTAL PROTOTYPE FOR AUTOMATICALLY *TESTING STUDENT PROGRAMS USING TOKEN PATTERNS**

Chung Man Tang, Yuen Tak Yu and Chung Keung Poon
Department of Computer Science, City University of Hong Kong, Tat Chee Avenue, Kowloon Tong, Hong Kong

Keywords: Automated Assessment Tool, Pattern-oriented Software Testing, Program Assessment Requirements, Program Validation, Testing of Student Programs, Token Pattern.

Abstract: Automated systems and tools for assessing student programs are now commonly used for enhancing the teaching and learning of computer programming. However, many such systems employ rudimentary techniques in comparing program outputs when testing student programs for determining their correctness. These comparison techniques are typically inflexible and disallow even slight deviations of program output which a human assessor would normally tolerate. This may give rise to student frustrations and other undesirable pedagogical issues that can undermine the benefits of using these assessment tools. This paper presents an experimental prototype we have developed that adopts a token-pattern-based approach to accommodate more tolerant output comparisons in testing student programs, followed by a preliminary validation of the prototype by showing how it can be configured to handle the assessment of variants of program outputs.

1 INTRODUCTION

The use of automated software systems and tools for assessing student programs is now popular in many universities (Ala-Mutka, 2005). These systems have not only relieved instructors' workload in administering and assessing student program submissions (Helmick, 2007; Joy et al., 2005), but also provided fast and useful feedback on students' work (Lam et al., 2008; Morris, 2003), thereby facilitating the use of enhanced pedagogy in Web-based or blended learning environments (Choy et al., 2007; Higgins et al., 2003), as well as increasing students' motivation to learn and improve through extensive practice (Law et al., 2010; Yu et al., 2006).

Ideally, all aspects of student programs should be evaluated to contribute to a holistic and impartial assessment, including but not limited to functional correctness (or simply *correctness*), run-time efficiency, memory usage, coding style and code structure (Ala-Mutka, 2005; Jackson and Usher, 1997). However, comprehensive assessment is very time-consuming and cannot be done very frequently for large classes. Moreover, each course may differ in its emphasis and value some aspects more highly than others. In most of the introductory programming classes, for instance, code correctness and structure is commonly considered more important than run-time efficiency.

One of the few aspects that are assessed automatically in most of the existing systems is the (functional) correctness of student programs, typically by means of testing (Jackson, 1991; Morris, 2003; Yu et al., 2006). A program is usually tested by executing it against a prescribed set of test cases and comparing its output (the *actual output*) in every test run with the *expected output* (that is, the output produced by a correct program with the input of the test run). Such a method of determining program correctness is known as the *(program) output comparison method* (Tang et al., 2009a).

This output comparison method has been used, in one form or another, in most of the existing systems, such as ASSYST (Jackson and Usher, 1997), BOSS (Joy et al., 2005), Ceilidh/CourseMarker (Higgins et al., 2003), HoGG (Morris, 2003) and PASS (Choy et al., 2008; Yu et al., 2006). In principle, the method can also be applied to non-text-based programs by

* The work described in this paper is supported in part by grants (project numbers 123206 and 123207) from the Research Grants Council of the Hong Kong Special Administrative Region, China.

Man Tang C., Tak Yu Y. and Keung Poon C. (2010).
AN EXPERIMENTAL PROTOTYPE FOR AUTOMATICALLY TESTING STUDENT PROGRAMS USING TOKEN PATTERNS.
In *Proceedings of the 2nd International Conference on Computer Supported Education*, pages 144-149
DOI: 10.5220/0002800301440149
Copyright © SciTePress

adding wrapper modules to convert their inputs and outputs into text strings (Morris, 2003). This paper, however, focuses mainly on program assessment in elementary programming courses, which in most cases require students to write text-based programs. Readers may refer to (Ala-Mutka, 2005) for a comprehensive survey of methods that apply to various types of non-text-based programs.

In practice, however, many existing systems employ rudimentary techniques in comparing the program outputs. These comparison techniques are typically inflexible and disallow even slight deviations of program output which a human assessor would normally tolerate (Jackson, 1991). This may give rise to student complaints, frustrations and other undesirable pedagogical issues that can undermine the benefits of such automatic assessment systems (Ala-Mutka, 2005; Higgins et al., 2003; Tang et al., 2009b; Yu et al., 2006).

In the rest of this paper, we shall first analyze the characteristics of common minor deviations of student program outputs (Section 2). We then review an improved output comparison approach based on token patterns (Section 3), present our newly developed experimental prototype which adopts the token pattern based approach to accommodate more tolerant output variants in testing student programs, followed by a preliminary validation using the prototype (Section 4). Finally, we conclude this paper in Section 5.

2 PROGRAM OUTPUT VARIANTS

The basic approach of implementing the output comparison method is to match the actual and expected outputs character by character (Ala-Mutka, 2005; Helmick, 2007; Jackson, 1991). In essence, this rudimentary character matching technique accepts the actual output as correct if and only if it is exactly the same text string as the expected output.

However, unless a programming exercise is prescribed with highly precise requirements and demands complete conformance, most instructors would agree that, for a given test input, there exist many variants of actual output that deviate "slightly" or "insignificantly" from the expected output (such as an extra blank space or fullstop at the end) but still can be accepted as correct when judged by a "reasonable" human assessor (Jackson, 1991). (For simplicity, in this paper, we shall refer to these outputs as *admissible variants*). Therefore, in practice, almost all current automated programming assessment systems supplement the basic character matching approach with some simple filtering strategies that disregard, for example, blanks, dots, hyphens or control characters (Ala-Mutka, 2005; Joy et al., 2005; Choy et al., 2008). These strategies are nevertheless ad hoc and still unsatisfactory.

To deal with this problem, we adopt a more fundamental and systematic approach. We first extracted a sample of programming exercise solutions previously submitted by our students across different courses, topics and intended learning outcomes. We then manually examined, in detail, the output variants that are rejected by our automatic assessment system as "wrong outputs" (Lam et al., 2008). Among these output variants, we were able distinguish between admissible variants (which we considered acceptable as correct) from other variants (which we considered as truly incorrect). Our analysis of the admissible variants showed that they are typically characterized as follows: (1) *typos*, such as misspelling of words, (2) *equivalent words*, that is, different words that basically have the same meaning as the expected output words, (3) *numeric precision*, that is, floating point numbers outputed in a higher or lower precision than the expected values, (4) *presentation*, such as spacing and relative positioning of output items, (5) *ordering*, which is regarded as immaterial for some programming problems, (6) *punctuation mark*, which is also considered immaterial for most programming problems. Note that this list is not exhaustive as it results from the analysis of our sample exercises only. It, however, serves as an useful guide for the design of an experimental prototype to improve the capability of automated systems in assessing admissible output variants.

Moreover, our analysis of the output variants makes it clear that most of the deviations pertain to information of elements such as words, numbers, ordering, etc., that cannot be easily captured by inspecting individual characters. Thus, instead of using a character-based matching approach, we experimented with an approach based on the notion of *token pattern*, recently introduced by Tang et al. (2009a). In the next section, we shall summarise the key notions involved and briefly describe the new approach of output comparison before presenting our experimental prototype in Section 4.

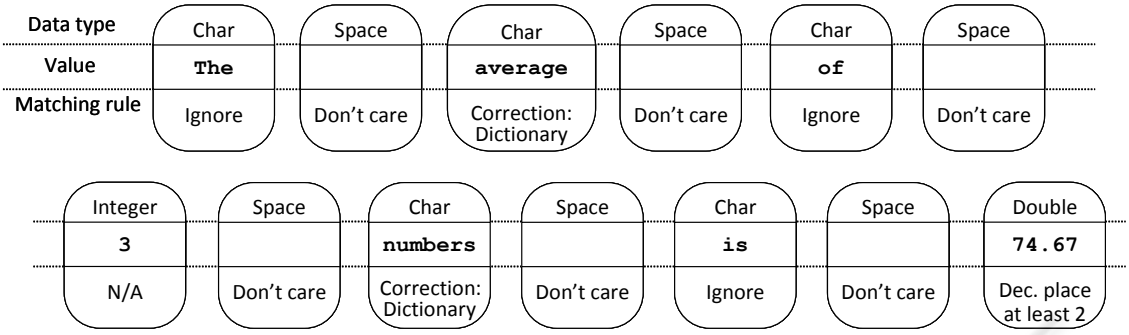


Figure 1: A token pattern example.

3 A TOKEN PATTERN BASED APPROACH

3.1 Token and Token Pattern

An output string can be decomposed into groups of successive characters, called *tokens*, representing meaningful pieces of information. To each token extracted from the expected output, one can attach precise criteria for comparison with tokens derived from an actual output. A *token pattern* thus refers to a string of tokens, each having a *data type*, *value*, and some associated (tagged) *matching rule*. For example, when an expected output token value is a floating point number, then the associated rule may state the desired minimum number of decimal places so that matching the token's value succeeds only if the actual output token has the same value, correct to the stated number of decimal places.

Figure 1 depicts an example token pattern converted from the output text: "`The average of 3 numbers is 74.67`". Here the blank "Space" separating the words and numbers are associated with a "Don't care" matching rule, meaning that the number of blank spaces is irrelevant as long as at least one is present as a separator. The stop words "`The`", "`of`" and "`is`" are specified as insignificant to be "Ignored" during matching. On the other hand, the "Char" strings "`average`" and "`numbers`" are significant, but variants are admissible if "Correction" can be made to match them using a built-in "Dictionary", thus allowing for minor deviations of equivalent words. For the "Integer" value "`3`", exact value matching is required and other rules are not applicable ("N/A"). Finally, the value "`74.67`" is of type "Double", and matching succeeds only if the other value agrees with it correct to "Decimal place at least 2".

3.2 Output Comparison based on Token Patterns

The token pattern approach works as follows. First, as usual, the instructor has to provide the expected output for each input. Next, the automated system splits the expected output string into a sequence of tokens. The system then automatically proposes some default matching rules for the tokens according to the type of token values and some configurable default options. The instructor can then fine tune the matching rules of individual tokens to determine exactly how the output tokens are to be matched.

Meanwhile, the automated assessment system also splits the actual output string into a sequence of tokens for matching with the expected output token pattern. A successful match according to the rules specified in the token pattern signifies that the actual output is acceptable as correct.

4 COMPARSION USING OUR EXPERIMENTAL PROTOTYPE

The token pattern approach was first conceptualised and proposed by Tang et al. (2009a). In this work, we have built an experimental prototype to validate the approach, evaluate its feasibility and explore the different possible options and matching rules.

Figure 2 shows a browser-based user interface of our experimental prototype for editing the default comparison options. In the figure, the specified options are as follows. (1) Character cases are insensitive (immaterial). (2) Stop words (in the given editable list) are ignored. (3) Other words are corrected by using both the Soundex algorithm (a phonetic algorithm which can correct minor deviations of words such as missing a vowel) and

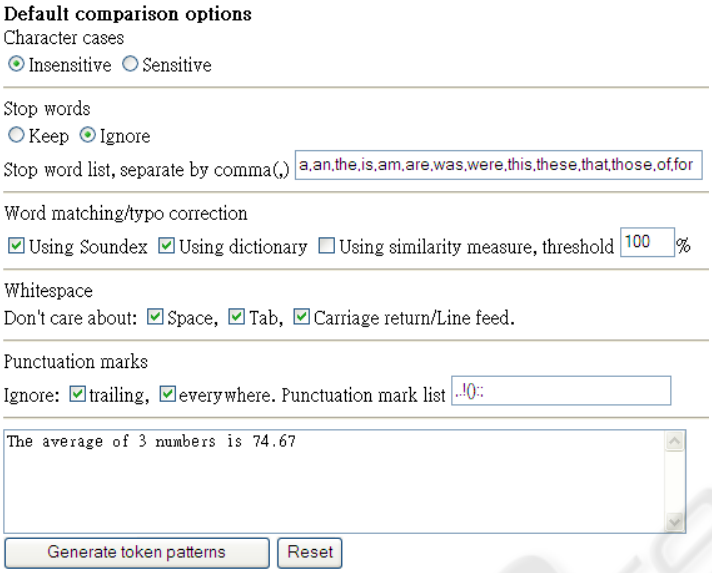


Figure 2: Interface of the prototype for editing the default comparison options.

searching for equivalent words using a built-in dictionary. (4) All whitespaces (including spaces, tabs, carriage returns/line feeds) are considered "Don't care". (5) All punctuation marks in the editable list are ignored, both at the trailing part of every line and everywhere else.

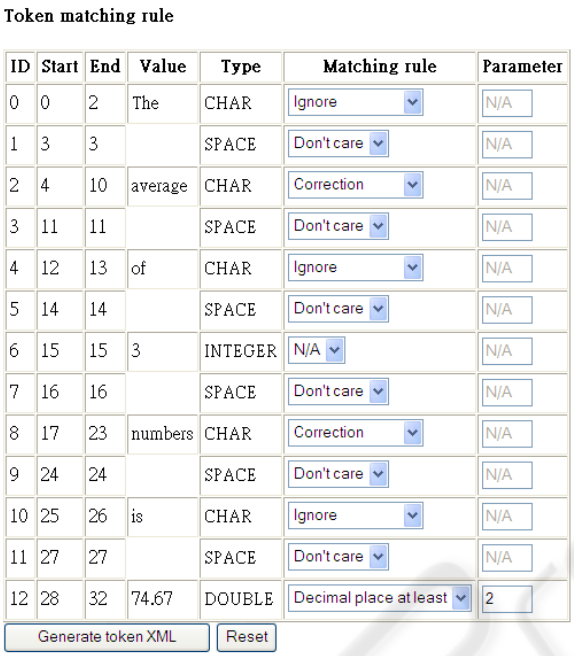
Next, the instructor provides the expected output (which may be automatically generated if a correct program exists, such as the one written by the instructor). The prototype automatically analyzes the expected output to generate a token pattern based on the output string and the specified default comparison options.

In the example shown in Figure 2, the expected output string is "`The average of 3 numbers is 74.67`". The string was decomposed by our prototype into 13 tokens, as tabulated in Figure 3. The table consists of 7 columns, namely, the token's ID, start and end positions in the output string, value, type, matching rule (automatically proposed by the prototype in accordance with the specified default comparison options) and its parameters, if any. Each row of the table corresponds to a token. The whole token pattern, which should now be self-explanatory, is the same as the one shown in Figure 1. Through the interface shown in Figure 3, the instructor may, if desired, fine tune the matching rule of any tokens at will. When editing of this table is completed, the prototype will generate an XML representation of the token patterns. Thus, once made, the instructor's choices are recorded in the XML representation to be subsequently interpreted by the prototype to perform the output comparisons accordingly.

The prototype thus allows the instructor to specify coarse-grained criteria (comparison options that apply to all tokens) as well as fine-grained criteria (matching rules), if desired, for individual tokens extracted from each test case. It also provides an easy-to-use and intuitively meaningful user interface to facilitate the specification of these criteria. Alternatively, the instructor can simply accept the default settings if they are considered appropriate for the programming exercise.

We now demonstrate the results of comparing two actual outputs with the token pattern specified in Figure 3. Figure 4 shows that (1) the word "`mean`" is accepted to be equivalent (or "considered matched") to "`average`" according to the built-in dictionary, (2) the singular word "`number`" is accepted as equivalent (or "considered matched") to its plural form while the extra colon "`:`" at the end of the word is ignored, and (3) the number "`74.6667`" agrees with "`74.67`", correct to 2 decimal places. Thus the actual output is acceptable according to the matching rules.

Figure 5 shows that, according to the Soundex algorithm, the misspelt word "`aevrage`" is acceptable, but not "`mumber`" (having a different sound). The latter is shown in red to indicate a mismatch. Thus, this output is rejected as incorrect.

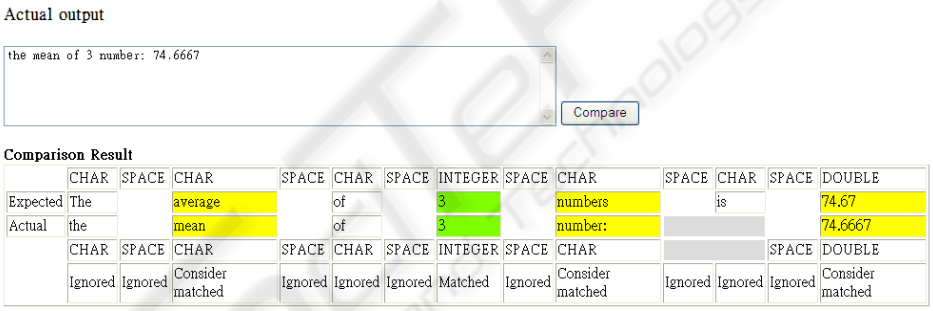


Token matching rule

ID	Start	End	Value	Type	Matching rule	Parameter
0	0	2	The	CHAR	Ignore	N/A
1	3	3		SPACE	Don't care	N/A
2	4	10	average	CHAR	Correction	N/A
3	11	11		SPACE	Don't care	N/A
4	12	13	of	CHAR	Ignore	N/A
5	14	14		SPACE	Don't care	N/A
6	15	15	3	INTEGER	N/A	N/A
7	16	16		SPACE	Don't care	N/A
8	17	23	numbers	CHAR	Correction	N/A
9	24	24		SPACE	Don't care	N/A
10	25	26	is	CHAR	Ignore	N/A
11	27	27		SPACE	Don't care	N/A
12	28	32	74.67	DOUBLE	Decimal place at least	2

Generate token XML | Reset

Figure 3: Interface of the prototype for editing the token matching rules.



Actual output

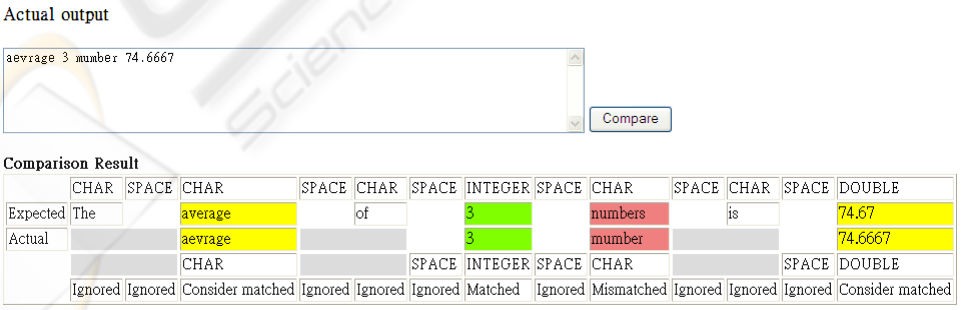
the mean of 3 number: 74.6667

Compare

Comparison Result

	CHAR	SPACE	CHAR	SPACE	CHAR	SPACE	INTEGER	SPACE	CHAR	SPACE	CHAR	SPACE	DOUBLE
Expected	The		average		of		3		numbers		is		74.67
Actual	the		mean		of		3		number:				74.6667
	CHAR	SPACE	CHAR	SPACE	CHAR	SPACE	INTEGER	SPACE	CHAR			SPACE	DOUBLE
	Ignored	Ignored	Consider matched	Ignored	Ignored	Ignored	Matched	Ignored	Consider matched	Ignored	Ignored	Ignored	Consider matched

Figure 4: An actual output of successful match.



Actual output

aevrage 3 number 74.6667

Compare

Comparison Result

	CHAR	SPACE	CHAR	SPACE	CHAR	SPACE	INTEGER	SPACE	CHAR	SPACE	CHAR	SPACE	DOUBLE
Expected	The		average		of		3		numbers		is		74.67
Actual			aevrage				3		number				74.6667
			CHAR			SPACE	INTEGER	SPACE	CHAR			SPACE	DOUBLE
	Ignored	Ignored	Consider matched	Ignored	Ignored	Ignored	Matched	Ignored	Mismatched	Ignored	Ignored	Ignored	Consider matched

Figure 5: An actual output rejected as incorrect.

5 CONCLUSIONS

Our experience, in common with the literature, is that students often feel frustrated with the strict output format requirements due to the automatic tester (Joy et al., 2005; Jackson, 1991; Morris, 2003; Yu et al., 2006). The inflexibility of existing output comparison approaches based on character matching gives rise to undesirable pedagogical issues that can undermine the benefits of using an automated assessment system (Tang et al., 2009b). To date, little progress has been made to address these (albeit well known) problems since the use of lex/yacc tools by Jackson (1991) and regular expressions (regex) in BOSS (Joy et al., 2005) and CourseMarker (Higgins et al., 2003). Nevertheless, regex and lex/yacc tools demand high proficiency of the user and are not reported to have been widely used in other systems.

In this work, we have characterized several common types of admissible variants and validated a newly proposed comparison approach based on token patterns by using an experimental prototype.

We recognize some limitations of our present prototype, such as the need for more matching rule options. Scope of this paper forbids detailed discussions on these. Fundamentally, however, the token pattern approach supports easy specification of matching criteria of varying granularity. It is intuitively easy to understand by various users. This is essential for determining adoption in practice and dealing with complaints. In time, we plan to identify areas of improvement of the new approach, perform experiments to evaluate its effectiveness, and assess the extent of additional effort required in practice.

REFERENCES

Ala-Mutka, K., 2005. A survey of automated assessment approaches for programming assignments. *Computer Science Education*, 15(2), 83–102.

Choy, M., Lam, S., Poon, C. K., Wang, F. L., Yu, Y. T., Yuen, L., 2008. Design and implementation of an automated system for assessment of computer programming assignments. In *Advances in Web-based Learning (LNCS 4823)*, 584–596. Springer.

Choy, M., Lam, S., Poon, C. K., Wang, F. L., Yu, Y. T., Yuen, L., 2007. Towards blended learning of computer programming supported by an automated system. In *Workshop on Blended Learning 2007*. Pearson: Prentice Hall.

Helmick, M. T., 2007. Interface-based programming assignments and automated grading of Java programs. In *12th Conference on Innovation and Technology in Computer Science Education (ITiCSE)*. ACM Press.

Higgins, C., Hergazy, T., Symeonidis, P., Tsinsifas, A., 2003. The CourseMarker CBA system: Improvements over Ceilidh. *Education and Information Technologies*, 8(3), 287–304.

Jackson, D., 1991. Using software tools to automate the assessment of student programs. *Computers & Education*, 17(2), 133–143.

Jackson, D., Usher, M., 1997. Grading student programs using ASSYST. In *SIGCSE'97, Technical Symposium on Computer Science Education*. ACM Press.

Joy, M., Griffiths, N., Royatt, R., 2005. The BOSS online submission and assessment system. *ACM Journal on Educational Resources in Computing*, 5(3), Article 2.

Lam, M. S. W., Chan, E. Y. K., Lee, V. C. S., Yu, Y. T., 2008. Designing an automatic debugging assistant for improving the learning of computer programming. In *ICHL 2008, International Conference on Hybrid Learning (LNCS 5169)*, 359–370. Springer.

Law, K. M. Y., Lee, V. C. S., Yu, Y. T., 2010. Learning motivation in e-learning facilitated computer programming courses. *Computers & Education*, in press.

Morris, D. S., 2003. Automatic grading of student's programming assignments: An interactive process and suite of programs. In *33rd ASEE/IEEE Frontiers in Education Conference*. IEEE Computer Society Press.

Tang, C. M., Yu, Y. T., Poon, C. K., 2009a. An approach towards automatic testing of student programs using token patterns. In *ICCE 2009, 17th International Conference on Computers in Education*.

Tang, C. M., Yu, Y. T., Poon, C. K., 2009b. Automated systems for testing student programs: Practical issues and requirements. In *SPECIAL 2009, International Workshop on Strategies for Practical Integration of Emerging and Contemporary Technologies in Assessment and Learning*.

Yu, Y. T., Choy, M. Y., Poon, C. K., 2006. Experiences with PASS: Developing and using a programming assignment assessment system. In *QSIC 2006, 6th International Conference on Quality Software*. IEEE Computer Society Press.