Teaching automated test case generation

Tsong Yueh Chen  
*Swinburne University of Technology*

Fei-Ching Kuo  
*Swinburne University of Technology, dkuo@uow.edu.au*

Zhiquan Zhou  
*University of Wollongong, zhiquan@uow.edu.au*

Follow this and additional works at: [https://ro.uow.edu.au/infopapers](https://ro.uow.edu.au/infopapers)

Part of the Physical Sciences and Mathematics Commons

**Recommended Citation**

Chen, Tsong Yueh; Kuo, Fei-Ching; and Zhou, Zhiquan: Teaching automated test case generation 2005, 327-332.  

Research Online is the open access institutional repository for the University of Wollongong. For further information contact the UOW Library: research-pubs@uow.edu.au
Teaching automated test case generation

Abstract
Software testing is a major approach to software quality assurance, but it is relatively neglected in universities' computing curricula. For students majoring in computer science or software engineering, several basic testing methods need to be taught. These testing methods generate test cases based on either specifications or program code. When introducing the testing methods based on program code, it is not easy to let the students experience automated test case generation due to the lack of supporting tools and limited teaching hours. In this paper we report our experience in teaching this topic with limited resources. The evaluation result indicates that our teaching method is effective and can also be adopted in other computer science/software engineering subjects where similar constraints exist.

Keywords
Teaching, Automated, Test, Case, Generation

Disciplines
Physical Sciences and Mathematics

Publication Details

This conference paper is available at Research Online: https://ro.uow.edu.au/infopapers/1588
Teaching Automated Test Case Generation

Tsong Yueh Chen  Fei-Ching Kuo ∗
Faculty of Information & Communication Technologies
Swinburne University of Technology
Hawthorn, Victoria 3122, Australia
{tchen,dkuo}@ict.swin.edu.au

Zhi Quan Zhou
School of Information Technology &
Computer Science
University of Wollongong
Wollongong, NSW 2522, Australia
zhiquan@uow.edu.au

Abstract

Software testing is a major approach to software quality assurance, but it is relatively neglected in universities’ computing curricula. For students majoring in computer science or software engineering, several basic testing methods need to be taught. These testing methods generate test cases based on either specifications or program code. When introducing the testing methods based on program code, it is not easy to let the students experience automated test case generation due to the lack of supporting tools and limited teaching hours. In this paper we report our experience in teaching this topic with limited resources. The evaluation result indicates that our teaching method is effective and can also be adopted in other computer science/software engineering subjects where similar constraints exist.

Keywords: Software engineering education, white-box testing, automated test case generation

1. Introduction

Software testing is a major approach to software quality assurance [5] and accounts for over 50% of the total software development cost. As pointed out in [10], however, the efforts and resources allocated to teaching testing in universities’ computing curricula are far from enough. With the ever-growing size and complexity of today’s software, there is an urgency of emphasizing the importance of testing and teaching useful testing methods to university students majoring in computing [2,3,7,8].

Any testing method belongs to either black-box or white-box category [1]. The former designs tests from the functional perspective and ignores the implementation details; the latter takes a structural point of view and generates test cases based on program code. Most white-box testing methods require certain features of the program code to be executed at least once. A basic rationale of this practice is that unless a portion of the code is executed, it is not possible for the bug uniquely associated with this portion to be revealed. The efficiency of white-box testing largely depends on the ability of automated test case generation. Throughout this paper we use the phrase “automated test case generation” to refer to automatically generating test cases to exercise prescribed paths or segments of the program code.

This paper reports our experience in teaching automated test case generation at Swinburne University of Technology (SUT), Melbourne, Australia. The organization is as follows. In Section 2 we describe the aims and practical constraints of teaching automated test case generation; in Section 3 we introduce the teaching method adopted to cope with the challenges; in Sections 4, we describe the two assignments, the major components through which the students learned and practiced automated test case generation techniques despite the limited teaching resources; in Section 5 we evaluate the effect of our teaching method and give some recommendations. Section 6 concludes the paper.

2. Constraints of Teaching Automated Test Case Generation

Software Testing and Reliability (HIT3057/8057) was a 12-week subject taught at SUT in 2004. Most of the students in the class were reading for a bachelor’s degree in Computer Science or Software Engineering, or a master’s degree (by course work) in Information Technology. To teach automated test case generation, there were two major constraints. First, the teaching hours were limited as the duration of the whole subject was only 12 weeks (2 lecture hours plus 1 tutorial hour per week), and automated test case generation was only one component of the subject. Although it is not difficult to introduce the basic concepts of various code coverage criteria, students need to learn much more in order to understand and implement automated test
case generation, which involves the following activities: (1) the construction of program control flow graphs, (2) the selection of paths to test in order to satisfy the coverage criteria, and (3) the generation of test cases that execute the selected paths. Because of the limited teaching hours, it was difficult for all these contents to be covered in the lectures. Secondly, because automated test case generation entails a lot of real work, the project would have been too large for this subject if the students had been asked to implement the whole automated test case generation system from scratch. Therefore we could only ask them to implement Activity (3) (which is regarded as the most difficult task in automated test case generation as it is equivalent to the halting problem [6]) with some supporting tools, such as program instrumentation and run-time monitoring software. ¹ This kind of tools (free software), however, were rare.

To meet the first challenge, that is, limited teaching hours, the teaching method of this subject was designed as “lectures plus tutorials plus literature reading plus implementation enforced by assignments”; to meet the second challenge, that is, the lack of supporting tools, we adopted a hybrid approach that was “simulative and partly manual”, with which the students could implement an automated test case generation method semi-automatically in their assignments. Details and effects of this teaching method will be reported in the following sections.

3. Design of the Teaching Method

Because the teaching hours were limited, the teaching method was designed as “lectures plus tutorials plus literature reading plus implementation enforced by assignments”.

In the lectures, basic concepts and principles of various coverage criteria were taught, such as statement coverage, branch coverage, condition coverage, path coverage, etc. The concept of path condition (that is, conjunction of branch predicates along the path) was also introduced. However, detailed procedures of automated test case generation were not covered in the lectures, such as how to solve constraints involved in path conditions or how to find actual input values to execute a prescribed path.

Every 2-hour lecture was accompanied by 1-hour tutorial. In the tutorials, the tutors elaborated on various steps involved in automated test case generation, such as how to construct control flow graphs and how to generate path conditions. Through the tutorials, the students learned that it involves a lot of time-consuming tasks to generate inputs that execute prescribed paths of the program code. Hence, they came to understand that it is necessary to automate the process as much as possible. Literature reading on automated test case generation, therefore, immediately followed as the first assignment to help the students obtain both broad and in-depth understanding of automated test case generation techniques. This assignment was further followed by Assignment 2 that asked the students to implement a particular test case generation method learned in Assignment 1.

It was expected that, through the combination of the lectures, tutorials, literature reading and implementation enforced by assignments, the students could obtain proper knowledge and experience of automated test case generation, and understand its major problems.

We appreciate the fact that, compared with sitting in the classroom and listening to the lecturers, students usually acquire a much deeper impression and a longer memory of things they learn on their own by going through and overcoming difficulties in the cause of learning. Furthermore, we hoped that the students’ abilities of independent learning could be enhanced through this training.

4. The Assignments

Two assignments were designed to help the students obtain both broad and in-depth understanding of automated test case generation techniques. To deepen their understanding, in the second assignment we asked the students to implement the method introduced in [9].

4.1. Assignment 1: Literature Reading

In Assignment 1, the students were required to read two papers, namely paper A [4] and paper B [9]. In fact, there had been many papers on automatic test case generation and it would be good if the students could read several of them. However, because it would take a long time for the students to read and understand them, we selected only two papers. The first is a survey paper [4] for the students to quickly establish a general overview of this area. It was expected that through reading this paper the students could understand basic principles, techniques and problems of automatically generating inputs that cover a selected part of the code. The second is a technical paper [9], which describes a particular method of automated test case generation. This paper demonstrates that although test case generation is equivalent to the halting problem, it can be reduced to a sequence of function minimization problems, and hence can be approached with various search algorithms. It was expected that after reading this paper the students could grasp this idea, acquire an in-depth understanding of Korel’s method and be able to implement it.

We prepared 8 questions in Assignment 1 to guide the students’ reading and help them capture the main ideas.
of the two papers. These questions can be classified into two categories regarding automated test case generation. Category 1 questions ask what has been done in the past as well as the basic concepts, principles and techniques; category 2 questions ask about some typical problems and how those problems could be addressed. Readers are referred to Appendix A for more details.

4.2. Assignment 2: Implementing an Automated Test Case Generation Method Using a Semi-Automatic Approach

In Assignment 2, the students were asked to implement the automated test case generation method introduced in paper B [9]. The purpose of this Assignment was to: (1) let the students practice an automated test case generation method through implementing it and using it to generate test cases for a real program, and (2) let the students experience how automated test case generation techniques can help reduce a tester’s workload.

In paper B, Korel proposed an automated test case generation method based on actual execution of the program under test. By means of program instrumentation, the execution flow of the program on an input can be monitored. If the intended path was not taken, it backtracks to the control statement where the wrong branch was taken. Then function minimization search algorithms are used to alter the values of the input variables automatically and alternately until the intended branch is taken. The basic search algorithm for function minimization in Korel’s method is called the alternating variable method. This process is repeated until the prescribed path is executed or it turns out that no progress can be made—in the latter case the search process fails to find a solution. The students learned the above method through the literature reading of paper B in Assignment 1.

In Assignment 2, students were required to implement the basic part (that is, the alternating variable method) of Korel’s method and generate test cases using this method for path coverage testing of a given program. For simplicity, the given program did not involve any loop or subroutine call. Infeasible paths were also excluded. Nevertheless, students were taught in the classes that infeasible paths widely exist in real life applications, and create a major difficulty in automated test case generation.

This assignment involved the following tasks: (1) construct the control flow graph; (2) identify the paths to test in order to satisfy the coverage criteria; (3) for each path, derive a test case to cover it using Korel’s method. Note that Korel’s method requires monitoring the execution flow, which in turn needs the instrumentation of the program code.

As mentioned in Section 2, due to the lack of supporting tools such as free software for program instrumentation, it would be impracticable for the students to fully implement (automate) Korel’s method. Therefore we adopted a hybrid approach that is “simulative and partly manual”. The students were required to complete tasks (1) and (2) manually as they were not difficult for the given program. For task (3), the students were required to write a program, namely ATCG (about 1000 lines of code in average, excluding header files), to simulate the monitoring of the execution flow of the given program and, when the input fails to execute the intended path, to alter the input values through searches and trials. As a result, while keeping the workload of the entire assignment moderate, the students were able to implement and experience the automated test case generation method. We would like to explain task (3) using the following simplified example, where G is the program under test, and x and y are its input variables.

\[
G(x, y) = \begin{cases} 
1: & \text{if } (2x > y - 3) \\
2: & x = x + 1; \\
3: & \text{if } (x + y < 2) \\
4: & y = x + y; \\
5: & \text{else} \\
6: & y = x - y; \\
7: & \text{return;} 
\end{cases}
\]

Suppose we want to generate a test case to execute the path \( p = (1, 2, 3, 4, 7) \). Students firstly have to manually identify the path condition for \( p \) to be traversed. (In this case, the path condition for \( p \) is \( (b_1) 2x > y - 3 \) and \( (b_2) x + y + 1 < 2 \) then prepare a file, say, input.txt, which stores an initial test case as well as all branch predicates associated with \( p \) \((b_1 \text{ and } b_2)\). The order of the predicates presented in input.txt reflects their order encountered in the control flow of \( p \). After that, the students need to pass input.txt to ATCG, which should then deduce the corresponding branch functions from \( b_1 \) and \( b_2 \): \( (f_1) -2x + y - 3 \) and \( (f_2) \) \( x + y - 1 \), and generate a test case for \( p \) according to Korel’s method. (Please note that in order that \( p \) can be traversed, \( f_1 \) has to be negative, before it is meaningful to have \( f_2 \) negative.) ATCG is supposed to simulate the program flow monitor by consistently evaluating the Boolean values of \( f_1 \leq 0 \) and \( f_2 < 0 \) every time \( G \) is executed with a test case. Although ATCG has all the branch functions, it should not solve these functions altogether. Once ATCG detects that the program flow starts diverging from \( p \) at a particular branch, ATCG should only apply the alternating variable method to that branch and make the corresponding branch function value \( < 0 \). A solution (a test case to traverse \( p \)) is found by ATCG when both \( f_1 \) and \( f_2 \) become negative.

Assignment 2 is outlined in Appendix B. To facilitate our examination of the students’ programs, in Task A we strictly defined the input and output format of their program as well as the details of the search steps for them to follow.
so that all their programs should produce identical outputs on a given input file.

4.3. Our Aid to the Students

To help the students understand the papers and the requirements of the assignments, we used the following approaches. Immediately after an assignment was released, we discussed the requirements with them in the tutorial. For Assignment 1, the students were allowed to discuss the papers in groups, but they were required to provide their own report on the assignment questions. Furthermore, in three tutorials, we used half of the session to let them discuss the papers and raise questions arising from the literature reading. For Assignment 2, the students were free to discuss their technical problems with each other and with the tutor in the class. After an assignment was marked, we spent one tutorial with them to address the problems found in their work.

5. Evaluation and Recommendations

Throughout the course we closely interacted with the students, both one-to-one and in their study groups, and carefully observed their feedback. Therefore we were able to see the effect of our teaching method on their learning. After the first couple of tutorials the students came to appreciate automated test case generation. Then through the literature reading, their knowledge on automated test case generation methods was broadened. Yet there was still vagueness in their minds after reading the papers, especially with regard to Korel’s method. Nevertheless, we observed that a lot of the vagueness had been clarified through the implementation of Korel’s method in Assignment 2.

We also noticed some problems by observing the students’ performance in Assignment 1. The students found that working through the questions and discussing questions in groups did help them a lot to understand the main ideas of the papers. On the other hand, because most of the students did not have much experience of reading academic papers, and also due to their heavy workload in other subjects, they felt it hard to digest the two papers within the required four-week time frame. As a result, we noted that the average marks of the class for the eight questions of Assignment 1 were all below 75% of the maximum mark of each question. In particular, they performed worst on questions 5 and 6 (please refer to Appendix A): for each of these two questions, about 40% of the students did not give a satisfactory answer. Question 5 asked about a relatively complicated technique used in Korel’s method. To answer this question, many students just copied the original texts from the paper word for word, which revealed their incomprehension of the technique. For question 6, compared to the other questions, it requires more complete understanding of Korel’s method and its details, which many students could not achieve within the required time frame; nevertheless, this problem was solved by doing Assignment 2. The above observations give us some hints on how to improve our teaching method in the future. We may need to provide the students with more helps and illustrations on the difficult parts of a technical paper. Furthermore, we may also require students to attend an oral assessment, so we can ensure that students do understand the technical content of Paper A and B rather than copying the text from the paper.

Figures 1 and 2 show the Assignment 1 and Assignment 2 marks of undergraduate and postgraduate students, respectively. The line of $Assignment\,2\,mark = Assignment\,1\,mark$ is dashed in the figures. The points above this line correspond to students whose performance in Assignment 2 was better than that in Assignment 1.

Figure 1 shows that out of the 24 undergraduate students who submitted the two assignments, 16 of them (that is, 2/3) demonstrated improved performance in Assignment 2. This implies that through reading two papers in Assignment 1, implementing Korel’s method in Task A and working on

---

2 Two points in Figure 1 overlap, so they are displayed as one point.
different activities in Task B, the students improved their understanding of automated test case generation.

Compared to the undergraduate students, the performance of the postgraduate students was very different. Figure 2 shows that, out of the 9 postgraduate students, 5 of them performed worse in Assignment 2 than they did in Assignment 1, and 6 of them received a mark below 50%. A further investigation on the students’ background revealed the reason: different from the undergraduate students, who were in software engineering or computer science major, most of the postgraduate students by course work were from other disciplines and, hence, they did not have a good programming skill. As programming tasks were weighted 70% of the total mark of Assignment 2, it was disadvantageous to them despite their relatively good performance on the non-programming activities (such as the manual construction of the control flow graph). This observation demonstrates that a prerequisite course in programming is necessary for this subject.

6. Conclusion

In this paper we have reported our experience of teaching automated test case generation. There were two major constraints on the teaching resources: limited teaching hours and a lack of supporting tools. We used a “lectures plus tutorials plus literature reading plus implementation enforced by assignments” teaching method to overcome the first limit, and adopted a hybrid approach that was “simulative and partly manual” to cope with the lack of supporting tools. In Assignment 1 the students conducted literature reading, and in Assignment 2 they implemented one of the automated test case generation methods learned through the literature reading. It was good to see that the students came to appreciate automated test case generation and acquired a broad and in-depth knowledge of it.

By closely interacting with the students throughout the course and observing their feedback as well as comparing their performance in the two assignments, it is evident that undergraduate students generally obtained better understanding of the theory after implementing it into practice; on the other hand, many postgraduate students did not perform well in Assignment 2 due to their limited programming experience. Despite that, we still see that the postgraduate students have benefited from Assignment 2, as they have gained practical experience of automated test case generation through their attempt to implement Korel’s method in Task A and conduct Activities 1-4 in Task B.

The effect of our teaching method and recommendations for improvement have been discussed in Section 5. Our experience reported in this paper is also transferable to other computer science and software engineering subjects where similar constraints exist.

A. Assignment 1 Outline

Category 1

Q1. “A path can be expressed as a set of equalities and inequalities (constraints)”. In the context of the papers, what does this sentence mean? In order to traverse a specific path, these constraints must be satisfied. Paper B uses the alternating variable method to search for test data. Explain how this can be done. Comment on its performance in complex program structures. The alternating variable method is a local search technique. Paper B states that local search techniques can easily stick on the local minimum, and hence global search technique may be used to search for test data. Give an example of global search techniques. Describe some possible drawbacks of this technique.

Q2. The techniques proposed in Paper B belong to the dynamic test data generation approach. What are the benefits and drawbacks of this approach?

Q3. Paper B states that the test data generation problem can be reduced to the function minimization problem. Explain how this can be done.

Q4. Discuss the role of the techniques of “influence network” and “constraint violation risk analysis” in Paper B.

Q5. Paper B proposes a method to generate test cases for programs that have structured inputs (such as trees). Explain how this can be done.

Q6. Provide pseudocode for the test data generation process in Paper B.

Category 2

Q7. Refer to Paper A. Explain what problems the existence of infeasible paths will bring to automatic control-flow coverage testing. Although determination of the existence of infeasible paths in a program is well known as an undecidable problem, it is desirable that we can detect infeasible paths in an early stage before these paths are passed to the test data generator. Describe some possible ways to detect infeasible paths.

Q8. When some of the program inputs are of non-numerical type, the techniques used in Papers B may not be applicable. Propose a method to deal with non-numerical inputs.

B. Assignment 2 Outline

Task A In this assignment, you are required to implement the alternating variable method described in Paper B. Your program should be written in C, C++ or Java. The input to your program consists of:

1. the number of input variables in the program under test;
2. an initial test case;
Intersect(), which is described in pseudocode as

\[
\text{Intersect}(x, \text{rec.min}_x, y, \text{rec.min}_y, x, \text{rec.max}_x, y) \text{ and } (\text{rec.min}_x, \text{rec.min}_y, x, \text{rec.max}_x, y),
\]

Intersect() returns a Boolean value to indicate whether the given rectangle and circle have an intersection. This function accepts 7 input variables: 

\[
\begin{align*}
\text{rec.min}_x, \text{rec.max}_x, \text{rec.min}_y, \text{rec.max}_y, \text{cir}_x, \text{cir}_y, \text{cir.rad}.
\end{align*}
\]

\text{cir.rad} = \text{radius of the circle. Y our task is to automatically generate test cases as the initial test case.

Task B You are required to test a module called Check\_Intersect(), which is described in pseudocode as shown in Figure 3. Check\_Intersect() returns a Boolean value to indicate whether the given rectangle and circle have an intersection. This function accepts 7 input variables:

\[
\begin{align*}
\text{rec.max}_x, \text{rec.max}_y, \text{rec.min}_x, \text{rec.min}_y, \text{cir}_x, \text{cir}_y, \text{cir.rad}.
\end{align*}
\]

\text{cir.rad} = \text{radius of the circle. Y our task is to automatically generate test cases as the initial test case.

Task B You are required to test a module called Check\_Intersect(), which is described in pseudocode as shown in Figure 3. Check\_Intersect() returns a Boolean value to indicate whether the given rectangle and circle have an intersection. This function accepts 7 input variables:

\[
\begin{align*}
\text{rec.max}_x, \text{rec.max}_y, \text{rec.min}_x, \text{rec.min}_y, \text{cir}_x, \text{cir}_y, \text{cir.rad}.
\end{align*}
\]

\text{cir.rad} = \text{radius of the circle. Y our task is to automatically generate test cases as the initial test case.

Activity 1: Construct the control flow graph for the Check\_Intersect() function.

Activity 2: List all paths in terms of their nodes. For each path, give its path condition.

Activity 3: Use the results generated in Activity 2 to create cinput.txt as the input to your program submitted for Task A. This file should contain an initial test case and all the path conditions of Check\_Intersect(). The file format is shown in Table 1. Use the first 7 digits of your student ID as the initial test case.

Activity 4: Run your program with your cinput.txt. Your program should generate test cases to achieve the path coverage for Check\_Intersect() function.

Table 1. Input file format

<table>
<thead>
<tr>
<th>Input</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>No. of input variables = 4</td>
</tr>
<tr>
<td>5.3 2 0 -7</td>
<td>The initial test case = (5.3, 2, 0, -7)</td>
</tr>
<tr>
<td>2</td>
<td>No. of paths = 2</td>
</tr>
<tr>
<td>3 - 6 5 -3 &lt; -22</td>
<td>No. of branch predicates for the 1st path = 2</td>
</tr>
<tr>
<td>6 - 8 9 6 &gt;= -53</td>
<td>No. of branch predicates for the 2nd path = 3</td>
</tr>
<tr>
<td>3 4 3 0 7 = 1</td>
<td>No. of branch predicates for the 3rd path = 1</td>
</tr>
<tr>
<td>-1 2 -2 1 &gt; -1</td>
<td>No. of branch predicates for the 4th path = 1</td>
</tr>
<tr>
<td>4 1 -4 3 &lt;= 10</td>
<td>No. of branch predicates for the 5th path = 1</td>
</tr>
<tr>
<td></td>
<td>No. of branch predicates for the 6th path = 1</td>
</tr>
<tr>
<td>Note: ABS(k) denotes the absolute value of k.</td>
<td></td>
</tr>
</tbody>
</table>

Figure 3. Pseudocode of Check\_Intersect()