5th World Congress for Software Quality – Shanghai, China – November 2011

Proposal of Execution Paths Indication Method for Integration Testing by Using an Automatic Visualization Tool 'Avis'

Yoshihiro Kita
University of Miyazaki
Miyazaki, Japan
kita@earth.cs.miyazaki-u.ac.jp

Tetsuro Katayama
University of Miyazaki
Miyazaki, Japan
kat@cs.miyazaki-u.ac.jp

Shigeyuki Tomita
University of Miyazaki
Miyazaki, Japan
stomita@cs.miyazaki-u.ac.jp

Abstract

One of the purposes of integration testing is to verify the validity of each module interface. It is important to verify the construction of programs by white-box testing but integration testing must deal with a massive increase in number of execution paths. This paper proposes execution paths indication method for integration testing that abstracts the execution paths covering all modules or all call-pairs, and reduces the number of execution paths by using an automatic visualization tool `called 'Avis' (**A**utomatic **Vis**ualization Tool for Programs). The execution paths indicated by this method can support the verification of data transfer by arguments and global variables of inter-modules. Moreover, the productivity of software testing is improved by reducing the number of execution paths.

1. Introduction

Integration testing is part of the software testing process, and its objectives include verifying the inter-module interfaces, completeness of functionality, and data manipulation by black-box testing. However, inter-module interfaces cannot be fully verified by black-box testing, because the validity of interfaces is related to the inter-modules structure.

To verify inter-modules interfaces, it is important to execute all modules and all call-pairs by white-box testing. However this is impossible due to the massive number of execution paths required to cover all modules and all call-pairs. It is also very difficult to manually select from these execution paths the necessary minimum for covering all methods and all call-pairs.

This paper proposes execution paths indication method for integration testing by using an automatic visualization tool `called 'Avis' (**A**utomatic **Vis**ualization Tool for Programs) for achieving this [1]. This method is abstraction and indication of the necessary minimum execution paths for covering all methods and call-pairs from the massive execution paths automatically.

We developed Avis to support the reading of Java programs for programming education [1]. Avis generates a flowchart which shows the flow of a program, and execution paths which show the behavior of the program from the source codes of a Java program as its input. The function of Avis in this proposed method is to indicate the behavior for all modules and all call-pairs of the program. Avis uses the concept of testing coverage for reducing the number of execution paths, and so we use Avis to support integration testing.

The execution paths generated by Avis include useless paths for supporting integration testing, because their paths are duplicated modules or call-pairs. We improve Avis to indicate the execution paths suitable for integration testing.

Section 2 describes the functions of Avis at present. Section 3 proposes improvements to Avis for integration testing. Section 4 evaluates the usefulness of Avis for indicating execution paths.

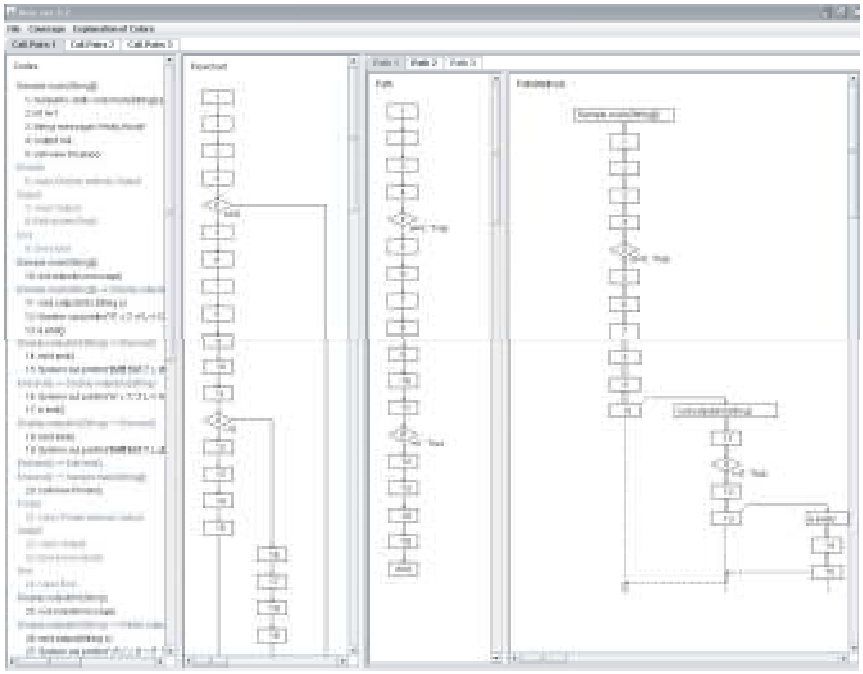

Figure 1. An output example of Avis

2. Existing functions of Avis

Figure 1 shows an output example of Avis. Avis generates a flowchart, sequential execution paths, and inter-modules execution paths automatically from the source codes of a Java program as its input.

2.1 Criteria for generating execution paths

Testing coverage exists to reduce the number of execution paths. Avis uses branch coverage (C1), which is a criterion indicating the rate of executed branches in all branches of a program. The execution paths generated by Avis satisfy C1.

The number of execution paths may increase to infinity if the program contains a loop, so we define the following two criteria based on C1 to ensure the number of paths is finite.

- Criterion for branch covering:
  All branches are executed at least once.

- Criterion for loop covering:
  All loops are iterated zero times (i.e. no iteration) and once.

2.2 Process of generating execution paths

Avis generates a flowchart from the source codes of a Java program, and generates the execution paths from the flowchart. It is obvious that the flowchart can be generated from the source codes of the program; the method of generating the flowchart is as described previously [2]. This paper describes a method of generating the execution paths that satisfy C1 from the flowchart, as shown in Figure 2. The process is as follows.

(1) Avis generates the flowchart from the source code of the programs (process (1) in Figure 2).
(2) Avis divides the flowchart according to the branch conditions (process (2) in Figure 2).
(3) Avis connects the divided codes and branches. The connections correspond to the execution paths (process (3) in Figure 2).

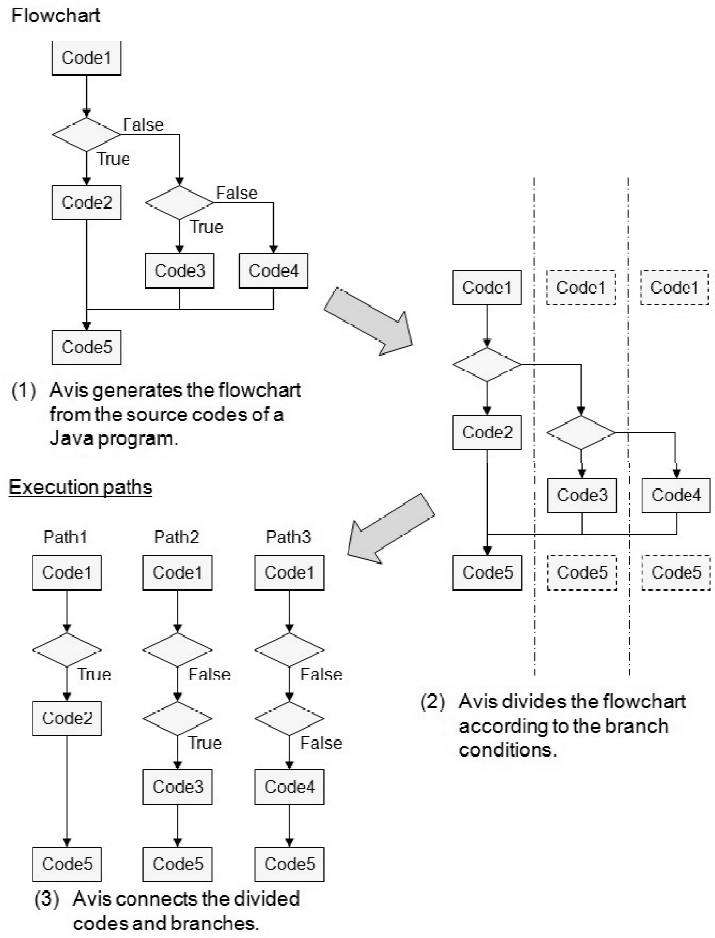

Figure 2. The process of generating execution paths from the flowchart

Avis can generate the execution paths covering all of the condition branches. Their paths satisfy C1. If a loop exists in the program, Avis divides the path into one iteration and no iteration to satisfy the criterion for loop covering.

3. Improvement of Avis for integration testing

Avis has been used as a visualization tool for programming education. We improve Avis to support integration testing.

3.1. Existing criteria for the integration testing

The following two criteria exist for testing coverage for integration testing [2]:

- Module coverage(S0)
  This means the proportion of executed modules to all modules of the program.
- Call-pair coverage(S1)
  This means the proportion of executed call-pairs to all call-pairs of the program.

These testing coverage are the completion criteria for integration testing. It is necessary to satisfy S0 or S1 in order to cover all modules or all call-pairs in the program. However, it is difficult to select the execution paths that satisfy S0 or S1, because the execution paths of each module are entangled intricately. Therefore, although the completed criteria for integration testing are called S0 or S1, there are few reports on methods of choosing execution paths for satisfying them.

3.2 Measures to obtain the relation among inter-modules

C1 is the test coverage for unit testing. It is difficult to generate execution paths that satisfy C1 if the program contains two or more modules. We have proposed a method of module unification by using inline expansion, which is an optimization technique used in research on compilers. Inline expansion involves spreading the codes of the called modules to the calling module [3]. Avis can generate execution paths that satisfy C1, because it can convert two or more modules to one large module.

Therefore, Avis can generate execution paths that cover all modules or all call-pairs, and satisfy C1 (i.e. Avis can generate execution paths that satisfy S0 or S1).

However, the number of execution paths may increase infinitely by the inline expansion if a recursive call exists in the program, so we define the following two criteria to limit the number of execution paths.

- Criterion for module covering:
  All modules are executed at least once.
- Criterion for recursive call covering:
  All recursive calls are iterated once.

3.3 Introduction of algorithm for abstracting execution paths

The generated paths that satisfy C1 include useless paths which duplicate the modules or call-pairs for treating these paths as execution paths satisfying S0 or S1. It is necessary to abstract from the generated paths those execution paths satisfy S0 or S1. We propose a mechanism for abstracting execution paths by using the 'stingy method' which is a method for solving combinational optimization in mathematical programming as follows.

(1) Select two paths from the execution paths that satisfy C1 and call each path 'cp1' and 'cp2'.
(2) Compare path 'cp1' with path 'cp2', and count the number of call-pairs in path 'cp1' which coincide with the call-pairs in path 'cp2'.
(3) If the number of all call-pairs in path 'cp1' is equal to the number counted by process (2), delete path 'cp1' and return to process (1).
(4) If the number of all call-pairs in path 'cp2' is equal to the number counted by process (2), delete path 'cp2', allocate another path to path 'cp2', and return to process (2).
(5) When processes (3) and (4) are not executed, another path which is not yet compared with path 'cp1' is allocated path 'cp1'. If all paths are compared with path 'cp1', path 'cp2' is allocated to path 'cp1' another path is allocated to path 'cp2', then return to process (2).
(6) When all paths have been compared with others, the remaining paths are 'the execution paths that satisfy S1'.
(7) Select two paths from the remaining paths in process (6), and call each path 'mp1' and 'mp2'.
(8) Check that the modules are included in each path 'mp1' and 'mp2'.
(9) If all modules included in the path 'mp1' are the subset of module included in the path 'mp2', delete the path 'mp1', and return to process (7).
(10) If all modules included in path 'mp2' are the subset of modules included in path 'mp1', delete path 'mp2', allocate another path to path 'mp2' and return to process (8).
(11) When processes (9) and (10) are not executed, another path which has not yet been compared with path 'mp1' is allocated path 'mp1'. If all paths are compared with path 'mp1', path 'mp2' is allocated to path 'mp1', another path is allocated to path 'mp2', then return to process (8).
(12) When all paths have been compared with others, the remaining paths are 'the execution paths that satisfy S0'.

We improved Avis so that this algorithm can be executed after generating the execution paths which is that satisfy C1.

3.4 Practicality of the execution paths

Avis analyzes statically the source codes of the program, and generates the execution paths according to branches regardless of the values of the conditional expressions. Therefore, Avis can indicate the execution paths before executing the integration testing. However, the program may be infeasible depending on the behavior of the execution paths. Infeasible paths may occur in the following two cases.

- Avis indicates execution paths that disregard the dependence between branch conditions.
- The execution paths include a part that is impossible to execute (called 'dead-codes').

It is difficult to judge the dependence between branch conditions by static syntax analysis only. Avis generates execution paths that can be executed based on syntax rules, so it is easy to find missed faults and unexpected flows by complicated branches such as multi-way branches.

Dead-codes are faults in the program even if they do not cause any problem during program execution. Users of Avis can find dead-codes in infeasible paths that contain dead-codes.

Therefore, Avis indicates infeasible paths because it can find the dead-codes in their paths.

4. Evaluation of the proposed method of abstracting execution paths by using Avis

We evaluate the proposed execution paths abstraction method by using Avis.

4.1 Evaluation of the scalability of Avis

We measure the runtime of Avis by using the following four programs as inputs to validate the scalability of Avis. Table 1 shows these runtimes.

- Program A (80 lines of code): Calendar
- Program B (155 lines of code): Simulation of bounding balls
- Program C (722 lines of code): Card game (Black jack)
- Program D (8,034 lines of code): Parser (a part of Avis)

The environment used for the measurement was 32-bit OS Windows Vista, twin Intel® Core™i7 2.93 GHz CPUs, and 4.00 GB of memory. Table 1 shows that the runtime of Program C (722 lines of code) is about 1 second, and the runtime of Program D (8,034 lines of code) is about 10 seconds. Thus, Avis can be executed for a large program in a practical time span.

4.2 Evaluation of the usefulness of reducing the paths by the proposed method

When using white-box testing for integration testing, all modules or all call-pairs must be executed at least once, in order to satisfy S0 or S1. The most positive method is that testers execute all modules or all call-pairs, but this approach is the worst in terms of performance. The number of execution paths resulting from this worst method is called 'the worst number'.

We compare the number of execution paths indicated by Avis to the worst number, and show the ratio of the reduction in number.

Table 1. Runtimes of Avis

Java program	Lines of code	Runtime(sec)
Program A	80	0.21
Program B	155	0.36
Program C	722	1.22
Program D	8,034	9.73

Table 2. Comparison between the worst numbers and the number of paths generated by Avis for satisfying S0 in each program

Java program (Number of modules)	Numbers of execution paths satisfying S0		Reduction ratio
	Worst number	Number of paths indicated by Avis	
Program A (7)	7	3	57.1%
Program B (23)	23	14	39.1%
Program C (76)	76	38	50.0%
Program D (573)	573	352	38.6%

Table 3. Comparison between the worst numbers and the number of paths generated by Avis for satisfying S1 in each program

Java program (Number of call-pairs)	Numbers of execution paths satisfying S1		Reduction ratio
	Worst number	Number of paths indicated by Avis	
Program A (17)	17	3	82.4%
Program B (66)	66	16	75.8%
Program C (259)	259	45	82.6%
Program D (2,277)	2,277	372	83.7%

Table 4. Number of executable paths among the execution paths satisfying S0 indicated by Avis

Java program	Number of execution paths indicated by Avis (Paths satisfying S0)…(1)	Number of executable paths…(2) (Ratio to paths of (1))	Number of modules covered by the paths of (2). (Ratio to all modules)
Program A	3	3 (100.0%)	7 (100.0%)
Program B	14	13 (92.0%)	22 (95.7%)
Program C	38	27 (71.1%)	61 (80.3%)
Program D	352	217 (61.6%)	409 (71.4%)

Table 5. Number of executable paths among the execution paths satisfying S1 indicated by Avis

Java program	Number of execution paths indicated by Avis (Paths are satisfying S1)…(3)	Number of executable paths…(4) (Ratio to paths of (3))	Number of call-pairs covered by the paths of (4). (Ratio to all call-pairs)
Program A	3	3 (100.0%)	17 (100.0%)
Program B	16	14 (87.5%)	63 (95.5%)
Program C	45	29 (64.4%)	217 (83.8%)
Program D	372	223 (59.9%)	1,558 (68.4%)

Table 6. Number of dead-codes included in the executable paths indicated by Avis

Java program	Number of infeasible paths included in the paths of (1) in Table 4…(5)	Number of dead-codes included in the paths of (5)	Number of infeasible paths included in the paths of (3) in Table 5…(6)	Number of dead-codes included in the paths of (6)
Program A	0	-	0	-
Program B	1	0	2	0
Program C	11	0	16	0
Program D	135	7	149	9

The ratio of reduction in the number of execution paths is given by the following expression.

$$\text{Reduction ratio of execution paths}(\%) = (\mathrm{W} - \mathrm{A}) \div \mathrm{W} \times 100$$

W: Worst number of execution paths that satisfy S0 or S1.
A: Number of execution paths indicated by Avis.

Avis can indicate the execution paths that cross two or more modules by unifying one module by the inline expansion. The number of paths is reduced because more modules or call-pairs are covered by only one execution. We confirmed the ratio of reduction in the number of execution paths by using Avis. The samples inputted to Avis were the four programs from section 4.1. Table 2 shows the reduction ratios of execution paths that satisfy S0, and Table 3 shows reduction ratios of execution paths that satisfy S1.

We confirm that the average reduction ratio of the execution paths that satisfies S0 is about 40% and the average that satisfies S1 is about 80%. Therefore, Avis successfully reduced the number of execution paths.

Table 4 shows the numbers of executable paths among the execution paths that satisfy S0. Table 5 shows the numbers that satisfy S1. Table 6 shows the numbers of dead-codes are included the infeasible paths indicated by Avis.

From Table 4 and Table 5, we confirm that the ratios of executable paths, and the numbers of modules and call-pairs covered by the executable paths, are reduced as the program becomes larger, and Table 6 confirms that the infeasible paths of a large program include dead-codes. Therefore, we can find the dead-codes in a program by using the infeasible paths generated by Avis.

4.3 Comparison of Avis with other integration testing tools

'Emma' [4] is a dynamic coverage measurement tool for white-box testing for both unit testing and integration testing. This tool indicates the execution paths of Java programs by highlighting the codes in three colors. Another tool called EclEmma [5] providing the functionality of Emma is offered as plug-in for Eclipse [6].

Although the users of Emma can obtain information on how many codes were executed since Emma measures the coverage by dynamic analysis, they do not receive information on how to execute the remaining paths, and how many codes remain in order to satisfy S0 or S1.

Avis indicates the execution paths that satisfy S0 or S1 irrespective of the condition of branches by static analysis. The users of Avis can grasp the essential execution paths for covering all modules or all call-pairs before performing integration testing. Avis also offers guidelines for generating test data by the indicated execution paths.

Methods [7] and [8] use integration testing tools by using dynamic methods based on the defined specifications. These tools generate executable test cases, and may solve infeasible paths. However, these methods cannot easily find dead-codes in programs because the users of these tools test only the executable parts of the programs as only the executable test cases are indicated.

Avis is useful because it indicates the infeasible paths by static analysis in order to find the dead-codes in programs.

5. Conclusion

In this paper, we have proposed a method for indicating execution paths for integration testing by using an automatic visualization tool `called 'Avis'. The method indicates the execution paths required to cover all modules or all call-pairs automatically, and reduces the number of execution paths.

We verified inter-module interfaces by using the execution paths for satisfying all methods or all call-pairs. We also confirmed that users of Avis can find dead-codes in programs by using the infeasible paths. Moreover, the proposed method assists efficient testing and can improve the productivity of software, because the execution paths are decreased and indicated before integration testing.

Future issues are as follows.

- Measures for infeasible paths
  There are two possible causes of existing infeasible paths: disregarding the dependence between condition branches, and the existence of dead-codes in programs. The infeasible paths of the former are useless paths, whereas those of the latter are paths that must be found. It is important to conduct semantic analysis in accordance with the syntax analysis for ascertaining the causes of existing infeasible paths.

- Applications of Avis to software testing education
  Avis has been used as a program visualization tool for programming education, but it can also be used to support software testing education so we need to improve Avis for integration testing. Therefore, it is necessary to consider the method of visualizing the execution paths as a requirement in software testing.

- Applications of Avis to other programming languages for integration testing
  Avis is currently for Java programs, but the proposed method could also be used for other modular programming languages. Avis should be improved to handle other programming languages.

References

[1] Kita, Y., Tokunaga, T., Katayama, T. and Tomita, S.: Extension and Evaluation of an Automatic Visualization Tool "Avis" for Programming Education, Proc. International Conferences on Software Engineering (SE 2009), as part of the 27th International Association of Science and Technique for Development (IASTED) International Multi-Conferences on Applied Informatics, pp.31–36 (2009).
[2] Pressman, S.R.: Software Engineering, 6th edition, The McGraw-Hill Companies Inc. (2005).
[3] Aho, V.A., Sethi, R. and Ullman, D.J.: Compilers, Addison-Wesley Publishers Limited (1986).
[4] SourceForge Inc.: EMMA code coverage,
available from http://sourceforge.net/projects/emma/ (accessed 2011-06-01)
[5] SourceForge Inc.: EclEmma – Java Code Coverage for Eclipse,
available from http://sourceforge.net/projects/eclemma/ (accessed 2011-06-01)
[6] Eclipse Foundation Inc.: Eclipse.org home. Available from http://www.eclipse.org/ (accessed 2010-06-09)
[7] Horrold, J.M. and Rothermel, G.: Performing Data Flow Testing on Classes, Proc. 2nd ACM SIGSOFT Symposium on the Foundations of Software Engineering, pp.154–163 (1994).
[8] Gallagher, L. and Offutt, J.: Test Sequence Generation for Integration Testing of Component Software, The Computer Journal of Oxford University Press on behalf of the British Computer Society (online), DOI:10.1093/comjnl/bxm093 (2007),
available from http://comjnl.oxfordjournals.org/cgi/reprint/bxm093v1 (accessed 2011-06-01)