Application of Equivalence Checking for Evaluation of Students' Programming Assignments

K. K. Sharma

Application of Equivalence Checking for Evaluation of Students' Programming Assignments

Thesis submitted in partial fulfillment
of the requirements for the award of the degree

of

Doctor of Philosophy

by

K. K. Sharma

Under the supervision of

Prof. Chittaranjan Mandal

Department of Computer Science and Engineering

Indian Institute of Technology Kharagpur

February 2019

APPROVAL OF THE VIVA-VOCE BOARD

Certified that the thesis entitled **“Application of Equivalence Checking for Evaluation of Students’ Programming Assignments”**, submitted by **K. K. Sharma** to the Indian Institute of Technology Kharagpur, for the award of the degree of Doctor of Philosophy has been accepted by the external examiners and that the student has successfully defended the thesis in the viva-voce examination held today.

Prof. Debasis Samanta
(Member of the DSC)

Prof. Arobinda Gupta
(Member of the DSC)

Prof. Chittaranjan Mandal
(Supervisor)

(External Examiner)

(Chairman, DSC)

Date:

CERTIFICATE

This is to certify that the thesis entitled **“Application of Equivalence Checking for Evaluation of Students’ Programming Assignments”**, submitted by **K. K. Sharma** to Indian Institute of Technology Kharagpur, is a record of bona fide research work under our supervision and we consider it worthy of consideration for the award of the degree of Doctor of Philosophy of the Institute.

Dr. Chittaranjan Mandal
Professor
CSE, IIT Kharagpur

Date:

Declaration

I certify that

a. The work contained in this thesis is original and has been done by myself under the general supervision of my supervisors.

b. The work has not been submitted to any other Institute for any degree or diploma.

c. I have followed the guidelines provided by the Institute in writing the thesis.

d. I have conformed to the norms and guidelines given in the Ethical Code of Conduct of the Institute.

e. Whenever I have used materials (data, theoretical analysis, and text) from other sources, I have given due credit to them by citing them in the text of the thesis and giving their details in the references.

f. Whenever I have quoted written materials from other sources, I have put them under quotation marks and given due credit to the sources by citing them and giving required details in the references.

K. K. Sharma

Acknowledgments

I take this opportunity to express my sincere thanks to my supervisor Prof. Chittaranjan Mandal for his valuable supervision, guidance, encouragement and advice throughout in this investigation. I would like to express my sincere gratitude to Prof. Dipankar Sarkar, earstwhile DSC member, for constant encouragement and evaluation of my research work.

At this point I would also like to thank Head of the Department of Computer Science & Engineering and the Director IIT Kharagpur for providing the necessary facilities during the entire period of research. I shall be ever indebted to Prof. P. K. Das, Dean PGS&R, for his kind support in thesis submission, without which this thesis would never get submitted. I also thank Prof. Arobinda Gupta and Prof. Debasis Samanta, the DSC members, for their fruitful suggestions. A special thanks to Prof. Pabitra Mitra, Prof. S. P. Pal, Prof. K. S. Rao, Prof. R. Mall, Prof. S. Misra, Prof. N. Ganguly, Prof. Shamik Sural, Prof. Abhijit Das, Prof. G. Biswas, Prof. Pallab Dasgupta, for their support in various ways. I would also like to thank Mr. Soumitri Mishra, Mr. Mithun Sarkar, Mr. Prasun Bhattacharya, Dr. Atanu Basu and all other staff members of Computer Science & Engineering Department, IIT Kharagpur for their needful help.

A special thanks to Prof. Manoj Tiwari, Prof. Pawan Kumar, Dr. Rajiv Misra, Dr. Tanmai Kulshreshtha, Mr. Arun Kumar Shukla, Dr. Jay Mant Jha, Dr. Rahul Das Gupta, Dr. Shashidhar Koolagudy, Dr. Prasenjit Bhavathankar, Prof. Mahesh Shirole, Dr. P. V. Rajkumar, Dr. Anant Nimkar, Prof. John C. John, Dr. C. Karfa, Dr. K. Banerjee, Dr. Rajendra Prasath, Prof. R. R. Suman, Indra Vikas, Late Prof. Vaibhav Tiwari, Ratan Kumar, Sathu Harish, Sudeep Kumar, Mukul Verma, Shri N. S. Debra and friends in Computer Science & Engineering Department and earstwhile SIT, IIT Kharagpur for their constant motivation and support. I also owe thanks to all those teachers and colleagues of SGSITS, Indore, whose motivation, encouragement and cooperation has helped me enormously in completing my research.

Last but not the least, I would like to take this opportunity to express my sincere gratitude to my parents Ram Rishi Sharma and Late Smt. Shanti Devi and siblings Late Rajeev, Late Lalit Kishor Sharma and Dr. Nawal Kishor Sharma and other relatives for their constant support.

K. K. Sharma

Abstract

This work develops a new method of program assessment based on equivalence checking of finite state machines with data-paths (FSMDs), reporting the error and providing feedback towards error correction. In order to assess a student's program, it is compared with a model program supplied by the teacher or instructor. For comparison, we actually establish the equivalence of their FSMDs, which is an intermediate representation of the programs and as FSMD is a graph like a flowchart, where the vertices are states and edges represent data and control flow, it provides scope for further analysis of paths in terms of statement containment, which is done after the execution of equivalence checker reports error. It uses containment checking algorithm, by which it categorizes the containment in the FSMD of a student's program into one of the three types of cases identified in this work, in order to assess the student's program. These categories of containment can be mapped to various types of errors in the programs with respect to the golden model, which may creep in while programming. The details of these errors and the methods for detecting and reporting them are described in this thesis. Various cases of errors in the students' programs have been correctly analyzed.

Conditional constructs require the conditions to follow a precedence order, so that the code corresponding to no condition is un-reachable. If there is a violation of precedence in occurrence of conditions, then the program can be immediately declared to be erroneous. This work describes how handling conditional constructs could be done as a pre-processing step to the equivalence checking to facilitate better feedback to the student. This work further addresses the variable mapping problem, as the reference program is likely to use variable names that are quite different to those used in the students' programs. This problem is important as the equivalence checking method requires that the names of the variables used must be the same, in the programs whose equivalence is to be established.

Next in the thesis, the automated checking of approximate equivalence of expressions, that cannot effectively be checked through formal equivalence checking, is described. It is based on a randomised simulation in a given domain. The final topic covered is an automated evaluation scheme for awarding marks to student programs.

Keywords: *FSMD, equivalence checker, containment checking, cut-point, last correct state (LCS), corresponding state of last correct state (CSLCS).*

Contents

Abstract **xiii**

Table of Contents **xv**

List of Symbols **xix**

List of Figures **xxi**

List of Tables **xxvii**

1 Introduction **1**

1.1 Introduction 1

1.2 Literature survey 3

1.2.1 Automated suggestions for correction of errors: survey of various approaches 7

1.2.2 Automated assessment of students' programs: survey leading to our formal method based approach 9

1.2.3 Related work for determining equivalence of two expressions 9

1.3 Motivation 10

1.3.1 Objective of the thesis 12

1.4 FSMD model 12

1.5 Normalization 26

1.6 Contributions of the thesis 29

1.6.1 A scheme for statement containment analysis of students' programs through equivalence checking 29

1.6.2 Methods to reconcile dissimilarities between FSMDs 30

1.6.3 Supporting techniques for checking and evaluation of students' programs 31

1.7 Thesis organization 34

1.8 Assumptions in the thesis 35

1.9 Conclusion 36

2 Containment analysis **39**

2.1 Introduction 39

2.2 Containment checking 40

2.2.1 Outline of containment checking algorithm 47
2.2.2 Interpretation of the results of containment checking 62
2.3 Implementations of strategies for various types of errors 68
2.3.1 Error of dependency violation 69
2.3.2 Reporting errors of parenthesis skipping 87
2.3.3 Error of missing block of code 98
2.3.4 Error of missing code in the nested cases 116
2.3.5 Missing the code of nested loop 117
2.3.6 Steps for correction mechanism 120
2.3.7 Simulation of correction mechanism on example problems . . 127
2.3.8 Missing code of nested condition checking 134
2.3.9 Steps for correction mechanism 137
2.3.10 Simulation of above mechanism 138
2.4 Summary of the strategies 142
2.5 Results and discussions 146
2.6 Conclusion 150

3 Methods to reconcile dissimilarities between FSMDs 151
3.1 Introduction 151
3.2 Programs with constraints in ordering of conditions 152
3.2.1 Identification and correction of precedence of conditions in `else-if` constructs 153
3.2.2 Implementation 156
3.2.3 Complexity analysis 162
3.2.4 Results 165
3.3 Variable mapping 167
3.3.1 Illustrative examples 168
3.3.2 Variable mapping algorithm 174
3.3.3 Demonstration of the algorithm 189
3.3.4 Various cases of variable mapping 191
3.3.5 Results 200
3.4 Conclusion 204

4 Supporting techniques for evaluation 207
4.1 Approximate equivalence checking of expressions 208
4.1.1 Introduction 208
4.1.2 Example of approximate equivalence checking 210
4.1.3 Equivalence checking with randomised simulations with some known properties 210
4.1.4 Obtaining the range of evaluation from the conditions in the program 224
4.1.5 The decision procedure 225
4.1.6 Results 227
4.2 Automated evaluation of programs 229
4.2.1 An automated program evaluation scheme 230
4.2.2 Results 235
4.2.3 Preliminary concepts of value propagation 235

4.2.4 Value propagation based automated program evaluation 237
4.2.5 An illustrative example 240
4.3 The benchmark programming assignment suite 241
4.3.1 Possible future extensions 245
4.3.2 Conclusion 246

5 Conclusion and future work 249
5.1 Summary of contributions 249
5.1.1 Containment analysis of students' programs through equivalence checking 250
5.1.2 Methods to reconcile dissimilarities between FSMDs arising from students' programs 251
5.1.3 Supporting techniques for checking and evaluation of students' programs 252
5.2 Future work 254
5.2.1 Enhancement of FSMD for pointers, I/O statements 255
5.2.2 Future work in variable mapping 261
5.2.3 Future work in approximate equivalence checking: Extension of containment checking by both ways path extension for approximate equivalence checking 261
5.2.4 Future work in FSMD to C conversion 263
5.2.5 Future work in debugging of evolving programs 264
5.2.6 Future work in debugging programs involving bit-vectors . . . 264

A Golden solution for each of the programming assignments 265

B Examples for handling cases of missing code of nested loops 271

C Outlines of a new procedure for correcting the student's programs 295
C.1 Introduction 295
C.2 Supporting procedures 296
C.3 Outline of an overall correction scheme 308

Bibliography 311

List of Symbols

M_i	Finite State Machine with Datapath (FSMD) of a program	12
Q_i	Set of control states of FSMD M_i	12
$q_{i,0}$	Reset state of FSMD M_i	12
I_i	Set of input variables of FSMD M_i	12
V_i	Set of storage variables of FSMD M_i	12
O_i	Set of output variables of FSMD M_i	13
τ_i	State transition function of FSMD M_i	13
$\mathcal{S}_i$	Set of status expressions of FSMD M_i	13
ν	Computation in an FSMD	18
$\nu_s \simeq \nu_g$	Computations ν_s and ν_g are equivalent	18
M_s	FSMD of student's program	19
M_g	FSMD of golden program	19
$M_s \sqsubseteq M_g$	FSMD M_s is contained in FSMD M_g	19
χ, ξ	Paths in FSMD	19
χ^s	Start state of path *chi*	19
χ^f	End (final) state of path χ	19
C_χ	Condition of execution of the path χ	20
r_χ	Data transformation of the path χ	20
s_χ	Storage variable transformations of the path χ	20
θ_χ	Output list of the path χ	20
P_1, P_2	Path cover of the FSMDs M_1 and M_2	20
$q_{s,i}$	A state of FSMD M_s	20
$q_{g,i}$	A state of FSMD M_g	20
$q_{i,j} \twoheadrightarrow q_{i,k}$	The path between the states $q_{i,j}$ and $q_{i,k}$	22
$q_{i,j} \overset{c}{\twoheadrightarrow} q_{i,k}$	The path between the states $q_{i,j}$ and $q_{i,k}$ having c as its condition of execution	22
$p_1 \bullet p_2$	path p_1 is concatenated with path p_2	42
$n \prec m$	n is ordered before m, the symbol used for precedence between elements with ordering	159
$x \mapsto y$	Variable x maps to variable y	169
$\mathcal{M}$	Map, the set of pairs of mapped variables between M_s and M_g	174

List of Figures

1.1 M_g: FSMD for golden program for computing GCD. 14

1.2 Figure explaining recursive definition of block structured FSMD. . . . 15

1.3 Figure explaining compaction of block structured FSMD. 16

1.4 Figure explaining compaction of block structured FSMD. 16

1.5 Figure explaining compaction of block structured FSMD. 17

1.6 Figure explaining compaction of block structured FSMD. 17

1.7 (a) M_g: Golden FSMD. (b) M_s: Student's FSMD. 23

1.8 Expression tree for method of obtaining normalization for $x+y*z-(c \div d)$. 28

1.9 Block diagram for automated assessment 34

2.1 FSMD of student's program 43

2.2 FSMD of golden program 43

2.3 M_g, path-wise one way contained example 45

2.4 M_s, path-wise one way contained example 45

2.5 M_s, path-wise un-contained example 47

2.6 M_g, path-wise un-contained example 47

2.7 M_s: path $q_c \rightarrow q_d$ shown with a dashed line, for which containment is to be found in M_g. 48

2.8 M_g: both ways containment of path $q_c \rightarrow q_d$ of M_s found in path $q_{02} \rightarrow q_{03}$, shown with a dashed line. 48

2.9 M_s: path $q_c \rightarrow q_f$ is shown with a dashed line, for which containment is to be found in M_g. 49

2.10 M_g: start with path $q_{02} \rightarrow q_{03}$, shown with a dashed line. 49

2.11 M_g: extend the path to q_{04}, shown with dashed lines. 49

2.12 M_g: finally containing path $q_{02}...q_{05}$ obtained, shown with dashed lines, by extending to q_{05}. 49

2.13 Parts of FSMDs (a) M_s and (b) M_g shown from LCS and CSLCS onwards. 53

2.14 Unordered path-wise both way contained Part-I: FSMD on fig (a) contains the dotted path which is equal to the dotted path shown in the fig (b). 62

2.15 Unordered path-wise both way contained Part-II: FSMD on fig (a) contains the dotted path which is equal to the dotted path shown in the fig (b). 63

2.16 Path-wise un-contained: FSMD in fig (a) contains the dotted paths which are not found in the fig (b) by containment checker. 64

2.17 Path-wise one way contained - I: FSMD in fig (a) show paths which are path-wise one way contained in the FSMD in the fig (b). 65

2.18 Path-wise one way contained - II: FSMD in fig (a) show paths which are path-wise one way contained in the FSMD in the fig (b). 65

2.19 Unordered path-wise both way contained and path-wise one way contained for faulty branching - I: FSMD in fig (a) shows the path, which is equal to the path shown in the FSMD in fig (b). 66

2.20 Unordered path-wise both way contained and path-wise one way contained for faulty branching - II: FSMD in fig (a) shows the path, which is contained in the path shown in the FSMD in fig (b). 67

2.21 Worst-case exponential nature of the containment checking algorithm. 68

2.22 Example of dependency violation. 72

2.23 Example of dependency violation: Working of equivalence checker I. 73

2.24 Example of dependency violation: Working of equivalence checker II. 74

2.25 Example of dependency violation: Working of equivalence checker III. 74

2.26 Example of dependency violation: Working of equivalence checker IV. 75

2.27 Example of dependency violation: Working of equivalence checker V. 75

2.28 Example of dependency violation: Working of equivalence checker VI. 76

2.29 Example of dependency violation: Working of equivalence checker VII. 77

2.30 Example of dependency violation: Working of equivalence checker VIII. 77

2.31 Example of dependency violation: Working of equivalence checker IX. 78

2.32 Example of dependency violation: Working of equivalence checker X. 78

2.33 Example of dependency violation: Working of equivalence checker XI. 79

2.34 Example of dependency violation: Working of equivalence checker XII. 79

2.35 Example of dependency violation: Working of equivalence checker XIII. 80

2.36 Example of dependency violation: Working of containment checker showing the unordered path-wise both way contained paths I. 81

2.37 Example of dependency violation: Working of containment checker showing the unordered path-wise both way contained paths II. 81

2.38 DAGs of (a) golden program and (b) student's program. 82

2.39 Dependency graph of golden program 83

2.40 Dependency graph of student's program 83

2.41 Dependency list of golden program 85

2.42 Dependency list of student's program 85

2.43 Case I: Figures showing (a) M_g and (b) M_s 89

2.44 Case I: Figures showing equivalent states in (a) M_g and (b) M_s 90

2.45 Case II: (a) M_g and (b) M_s 91

2.46 Case II: Equivalence found for the dark edges but not for the dotted edges of M_s 91

2.47 Figures showing the output of algorithm for case II. Dark edges are the entry present in stack1 and stack2. 96
2.48 Figures showing (a) M_g and (b) extension of loop in M_s due to skipping parenthesis. 97
2.49 Figures showing (a) M_g and (b) M_s, showing equivalence checker failing for path shown with dark line 98
2.50 M_g and M_s, M_s has a missing block of code 101
2.51 M_g and M_s, M_s has a missing block of code 102
2.52 M_g and M_s, M_s has a missing block of code 102
2.53 M_g and M_s, M_s has extra block of code 103
2.54 M_g and M_s, M_s has a missing block and an extra block of code 104
2.55 M_g and M_s, removing code from extra block in M_s 104
2.56 M_g and M_s, merging the states at the two ends of extra block in M_s . . 105
2.57 M_g and M_s reversed, equivalence check finds $M_g \not\sqsubseteq M_s$ 105
2.58 Figure showing M_g with several blocks of code. 108
2.59 Figure showing M_s with missing block of code. 109
2.60 Figure showing M_s with application of correction vector. 110
2.61 Figure showing M_s with application of correction vector for the second time. 111
2.62 Figure showing M_s becoming same as M_g after correction done for skipping parenthesis. 113
2.63 An M_g for digitsum program 118
2.64 An M_s for digitsum program. 119
2.65 Outgoing transitions from loop state 121
2.66 M_g, generate tables program 128
2.67 M_s, generate tables program 128
2.68 Modified incorrect FSMD after introducing loop1 before the unmatched state in M_s for generate tables program. 130
2.69 Modified incorrect FSMD after introducing loop2 inside loop1 in M_s for generate tables program. 132
2.70 Modified incorrect FSMD after introducing chain in loop 2 in M_s for generate tables program. 133
2.71 M_g having nested conditions. 135
2.72 M_s missing nested conditions. 136
2.73 After introduction of missing state q'_{06} in M_s. 139
2.74 After introduction of missing states q'_{07}, q'_{08} and q'_{19} in M_s. 140
2.75 Corrected M_s, after introduction of missing states q'_{09} and q'_{20}. 141

3.1 Depicting a normalized cell 161
3.2 Depicting the expression 3 + 2 * a + 5 * x * y with list of normalized cells 162
3.3 An example of case 1 165
3.4 Another example of case 1 165
3.5 An example of case 2 165
3.6 Another example of case 2 165
3.7 An example of case 3 166
3.8 Another example of case 3 166

3.9 An example of case 4 166
3.10 Another example of case 4 166
3.11 An example of case 5 166
3.12 Another example of case 5 166
3.13 An example of case 6 166
3.14 Another example of case 6 166
3.15 FSMD 1 of example 3.1 169
3.16 FSMD 2 of example 3.1 169
3.17 FSMD 1 of example 3.2 170
3.18 FSMD 2 of example 3.2 170
3.19 Two FSMDs for example 3.3 172
3.20 Two FSMDs: demonstration of variable mapping program 189
3.21 Two FSMDs: Case I, variable mapping 192
3.22 FSMD 1: Case II, variable mapping 196
3.23 FSMD 2: Case II, variable mapping 197
3.24 Two FSMDs: Case III, variable mapping 199

4.1 FSMDs (a) M_s and (b) M_g. 217
4.2 FSMDs (a) M_s and (b) M_g. 219
4.3 FSMDs (a) M_s and (b) M_g. 219
4.4 An example of value propagation. 236
4.5 (a) Model FSMD, M_g. (b) Student FSMD, M_s for example 4.12. . . . 238
4.6 An example to illustrate automated program evaluation scheme. (a) Model FSMD, M_g. (b) Student FSMD, M_s. 240

5.1 Parse tree for `&v` 258
5.2 Data Structure used for NCell 259
5.3 Data Structure used for read/ write 261
5.4 FSMDs (a) M_s and (b) M_g. 262

B.1 M_g, digitsum program 272
B.2 M_s, digitsum program 272
B.3 Modified incorrect FSMD after introducing loop1 before the unmatched state in M_s for digitsum program. 274
B.4 Incorrect FSMD after modifying loop1 in M_s for digitsum program. . 276
B.5 Modified incorrect FSMD after introducing chain in loop2 in M_s for digitsum program. 278
B.6 Modified incorrect FSMD after completion of chain copy in loop1 in M_s for digitsum program. 280
B.7 M_g, diamond program 281
B.8 M_s, diamond program 281
B.9 Modified incorrect FSMD after introducing loop1 before the unmatched state in M_s for diamond program. 283
B.10 Incorrect FSMD after inserting loop2 in M_s for diamond program. . . 284
B.11 Modified incorrect FSMD after introducing chain in loop2 in M_s for diamond program. 285
B.12 M_g, a simple program 286

B.13 M_s, a simple program 286
B.14 M_g, digitsum program 289
B.15 M_s, digitsum program 289
B.16 Modified incorrect FSMD after introducing loop1 before the unmatched state in M_s for digitsum program. 291
B.17 Incorrect FSMD after modifying loop1 in M_s for digitsum program. . 293

C.1 FSMD of Golden Program, in the figure `t0..t11` are the set of statements, e.g. `t0={a=5;b=4;a=b+c;..}` 298
C.2 Figure with FSMDs for various examples. 302
C.3 Figure with FSMDs for finding similarity and overll correction. 310

List of Tables

2.1 Table of cases of error diagnosis of program FSMDs 148
2.2 Table of cases of error handling of program FSMDs for missing nested or in-line code 149

3.1 Examples for `else-if` constructs 156
3.2 Extended examples for `else-if` constructs 157
3.3 Results for `else-if` constructs 167
3.4 Condition not matched 200
3.5 More variables 201
3.6 Different output 201
3.7 Wrong output 202
3.8 Missing loop 202
3.9 Uninitialized variables 203

4.1 Checking expression equivalence 229
4.2 Automatic evaluation vs TA's evaluation for Coordinate.c, C1=3, C2=7 231
4.3 Automatic evaluation vs TA's evaluation for Friend.c, C1=3, C2=7 . . 231
4.4 Automatic evaluation vs TA's evaluation for Equal0_1.c, C1=3, C2=7 232
4.5 Automatic evaluation vs TA's evaluation for GCDBenchmark.c, C1=3, C2=7 232
4.6 Automatic evaluation vs TA's evaluation for LCM.c, C1=3, C2=7 . . . 233
4.7 Automatic evaluation vs TA's evaluation for GCD.c, C1=3, C2=7 . . . 233
4.8 Automatic evaluation vs TA's evaluation for BarCode.c, C1=3, C2=7 . 234
4.9 Automatic evaluation vs TA's evaluation for Swap.c, C1=3, C2=7 . . . 234

A.1 Table for comparison of various features of benchmark programs . . . 270

C.1 Dominator table 300

Chapter 1

Introduction and literature survey

1.1 Introduction

Computers are being used heavily to aid learning. Students can be taught and evaluated through computers. An automated tutoring system and automated evaluation system are needed for automated electronic learning. This thesis describes principles of the design of a tool for automated evaluation of students' programs.

Automatic evaluation is considered to be consistent and fast, as compared to human evaluation, which is not always consistent, let alone being fast.

We now introduce briefly our scheme for automated evaluation of students' programs. A program can be visualized as a finite state machine with data-path (FSMD). Automated evaluation can be aimed at by using equivalence checking of FSMDs of two programs, which is a well studied problem [44, 45, 47, 54]. In our approach, for a given programming assignment, the instructor is expected to provide a golden solution to serve as the source of a reference FSMD. The program of the student serves as the source of the other FSMD. For our approach we leverage equivalence checking techniques for two FSMDs pioneered locally [44, 45, 47, 75]. The bare mechanism for equivalence checking is to identify a finite set of path segments to capture any computation in one of the FSMDs. Thereafter, a set of corresponding path segments are identified in the other FSMD, so that for any computation expressed as a concatenation of path segments in the first FSMD, there is a corresponding computation in the second FSMD, which can be expressed as a concatenation of corresponding path segments in the other FSMD.

In the basic equivalence checking scheme, the objective would be only to determine whether or not the two programs are equivalent. For this work the objective is broader. We also seek to understand how the student program deviates from the golden solution and use that to “correct” the faulty student program and explain the correction applied. That way our scheme is expected to be an automated assist mechanism to aid the student to write correct programs. We do not seek to correct syntax errors; it is assumed that the given program is syntactically correct. Also, a problem may have multiple solutions whose equivalence may be non-trivial to prove automatically. This is addressed by using a set of golden programs rather than a single program. The golden program that is found to be the “closest” to the student program is considered as the reference. With the wide relevance of static pedagogical content available on the Internet, it is becoming increasingly important to have dynamic mechanisms to aid teaching. An area that deserves attending to is the application of formal methods to assist evaluation of content. In this work we focus on the application of equivalence checking techniques to assist student program evaluation for correctness. We consider this to be an important problem because programming is considered to be a basic engineering and scientific ability, yet evaluating programming exercises is a skill intensive exercise, of which there is a short supply of experts. This claim is based on the student-teacher ratio in a typical course involving programming, where a single teacher may have to deal with 100 or more number of students. The overall goal is lofty and we have made only a humble beginning in this direction.

This method has potential application in automatically carrying out evaluation of programs in an environment where there is a large number of students, thus saving a lot of time for the instructor. Being automated, our scheme will ensure consistency in evaluation and ensure speedy evaluation. The major contributions of this work are: *i)* extension of equivalence checking method of FSMDs, by checking statement containment, for the diagnosis of errors in student programs, *ii)* identification of classes of some errors in programs, *iii)* detecting violation of precedence in if-else-if constructs, in a student's program *iv)* variable mapping for the preprocessing of student programs, *v)* approximate equivalence checking of mathematical expressions, *vi)* suggesting a scheme for automated marking of student programs. This work can be used in many fields, not only in assessment of a student's ability of computer programming but also for detecting some potential errors in computer programs, if an “ideally” and correctly developed program exists.

1.2 Literature survey

Literature review papers have been published on automated assessment in 2005 by Mutka [7] and the subsequent developments from 2006 to 2010 have been reported by Ihantola et al. [38]. Liang et al. [55] have also reported a survey of automated programming assessment tools.

Douce et al. [27] in their review have classified automated assessment tools in terms of three generations, viz, early assessment systems, tool-oriented systems and web-oriented systems. Automated assessment techniques have been developed for judging partial correctness of programs using graph based similarity [64]. Some of the features in the automated assessment systems have been the test definitions for the black-box or white box testing, resubmission policies and sand-boxing issues for secure submission [58]. Model checking based approaches for checking programs are available in the literature, some of which are by Clarke, Kroening & Lerda [24], Yu, Duan, Tian & Yang [94], Merz, Falke & Sinz [61] and Rocha, Ismail, Cordeiro & Barreto [70] etc. In these, model checking has been used for property checking of programs. The method of Clarke, Kroening & Lerda [24] handles the programs which can be unwound. In model checking, there is a model, which is a finite state system. A program, whereas, is not necessarily a finite state system as far as data values are concerned (if true integers are considered), resulting in infinite data states. The number of control states is finite. In case of model checking, the objective is to determine whether a given model satisfies a given temporal property. State space explosion is a commonly encountered problem in model checking. Bounded model checking [17, 61] is a restricted form of model checking for a given extent of loop unwinding. In case of equivalence checking we want to ensure that the transformation of data values is also consistent, which is not captured by model checking. Equivalence checking considers the equivalence of computations performed by two systems under consideration, one of which could be a reference model. Assessment of a student program is aligned to equivalence checking where conformity with a golden program is considered.

There are many approaches to designing the systems for automatic assessment of students' programs. In the literature there are three major approaches - dynamic analysis, the use of software metrics and static analysis [91]. In the survey by Ala Mutka [7], which is a well cited paper on survey of automated assessment approaches for programming assignments, two broad categories of approaches have been discussed for automated assessment, dynamic and static. A brief detail of what they have discussed in [91] about the methods is as follows. In dynamic analysis the student program is executed and the output is checked for correctness. In case of dynamic approach, where the student's program is to be executed for assessment, there is an essential requirement to provide a secured running environment like a sandbox, for running students' programs without risks to the surrounding environment as the student's program may have bugs or it may be designed intentionally to be malicious. In dynamic approach a program is tested for its *i)* functionality and *ii)* efficiency.

Also, in this approach, the students are supposed to write programs which pass given test cases. Some examples of tools for functionality checking are Ceilidh [15] (now CourseMarker [37]), Assyst [39], Homework Generation and Grading project (HoGG) at Rutgers University [63], Online Judge [21], and BOSS [40]. Some examples of tools for checking efficiency are Ceilidh [15] (now CourseMarker [37]), Assyst [39] and Online Judge [21]. Some special features that have been incorporated in tools using dynamic approach are e.g., *i)* a feature to allow assessment in the middle of processing, defining a test case with a planned relationship to the program state created during previous test input. Quiver uses a similar approach. *ii)* language specific implementation issues, e.g., testing dynamic memory management in C++ language. These systems, such as TRY [69], Scheme-robo [74], Online Judge [21], Quiver [29], and RoboProf [26], usually adopt a dynamic testing assessment mechanism which runs a student program through a set of testing data. The only factor that can influence the evaluation is the success or otherwise of a test [90]. Further to the drawbacks of the dynamic testing assessment mechanism, the following has been observed in [90]: *" ... does not take into account the way in which a problem has been solved. This sometimes leads to inequitable grading results, because a program producing a right output may not meet the programming specification. There is another drawback of dynamic testing based assessment systems. In many cases, a program, submitted by a novice in an examination, may not produce an output or even may not terminate when dynamically tested, which makes these systems fail to give reasonable marks".*

Further we quote from [90]: "CourseMarker [37], Boss [40], and GAME [18, 19], improved the dynamic testing assessment mechanism by introducing static analysis. These tools perform quality or style checks against a set of metric tools and provide feedback. GAME also performs structure analysis by examining strategic key points in the code. However, the drawbacks of the dynamic testing assessment mechanism have not been well solved yet".

In other methods software metrics such as lines of code, number of variables, statements and expressions are used for the assessment.

The other approach discussed by Mutka [7] is static assessment, where evaluation can be carried out by collecting information from program code without executing it. In static analysis the student program is not executed but it is compared to some standard program usually on the basis of its structure. The extent of similarity between the two gives an estimate for the assessment. Although commonly assessment is done by executing the submitted programs, the static assessment has its own advantages like *i)* exposing the functionality features that may remain unnoticed by the limited set of test cases, *ii)* can be used even if the program is buggy or has malicious intent. Most static assessment methods rely on the formal structure of the program and thus they require the program to be syntactically and semantically correct.

The static method is used to assess the following features: Coding style, programming errors, software metrics, design. Search for some word or expression in program code and plagiarism detection features are among the special features incorporated in tools using static method.

In this survey the emphasis is more on the static assessment methods. In all the static methods that we came across, the student program is compared against a standard program chosen from a set of model solution programs. Both the programs that are to be compared are at first transformed and normalized in some way and thus converted to some intermediate representation so as to become more worthy of comparison (i.e., easier to compare). In most of the cases the intermediate representation is either some graphical representation e.g. a system dependence graph (SDG) [91], an abstract syntax tree (AST), an augmented object-oriented program dependence graph (AOPDG), or it may be a textual representation e.g. Backus Naur Form (BNF). Before comparing, both the programs are converted to some intermediate representation in order to capture their control and data-flow graphs explicitly, for example, the work reported in [91] extracts system dependence graphs from both the student and the instructor supplied programs before checking for equivalence.

Many researchers point out that automatically detecting equivalence between two arbitrary programs or two mutants is an undecidable problem; however, for many specific cases, equivalence can be decided [91].

In the context of automatic grading, student programs and model programs are much simple, and they may be alike, so we can determine their equivalence and calculate their semantic similarities [91]. Songwen Xu et al. [93] have described in details a transformation based approach to automate the diagnosis of students' programs.

A grading framework combining the approaches of testing, software verification, and control flow graph similarity measurement have been reported in [89]. The tools make use of an intermediate code representation, thus being applicable to various languages.

The static method has its own advantages and hence dynamic systems were improved by the use of static method. In recent systems like AutoLEP [90] and eGrader [76], a combination of both static and dynamic approaches is visible.

A pseudo-code comparison technique is reported in the work by Rahman et al. in [68] and its evaluation discussed in [67].

A service-oriented approach for the design and implementation of an automatic assessment system for programming assignments has been reported in [9] by M. Amelung et al.

In some studies unit testing has been used for automated assessment tools. The unit testing is defined as follows: "In computer programming, unit testing is a software testing method by which individual units of source code, sets of one or more computer program modules together with associated control data, usage procedures, and operating procedures, are tested to determine whether they are fit for use." [50]. Use of JUnit, a unit testing framework for Java, has been made for designing automated assessment tools for programming exercises in Java. BOSS [40] and Oto [86] are examples of such a tool using JUnit. A marking tool should incorporate the following aspects viz., to assess the correctness of programs, quality of the code, efficiency of the program, quality of the tests developed by the students to test their own program and ability to detect originality of the code i.e., plagiarism detection.

Recently a graph-based grading system for introductory Java programming has been introduced by Shamsi et al. [76]. This system, called eGrader, grades submission both dynamically and statically. While dynamic analysis is based on JUnit framework, the static analysis is based on the graph representation of the program and its quality which is measured by software metrics. The graph representation is based on the Control Dependence Graphs (CDG) and Method Call Dependencies (MCD), which are constructed from the abstract syntax tree of the source code They have also introduced a notion of Identification pattern, which is an encoded pattern of digits, to analyze both the structure and the Software Engineering Metrics (SEM) of students' programs. Identification pattern matching is based on the distance between them. The distance measure has been defined as the number of missing control structures and SEM components from the model program in addition to the number of extra control structures and SEM component in the students' identification pattern. Formally, it is: $D = |N_{Missing} + N_{Extra}|$, where D is the distance, $N_{Missing}$ is the number of missing control structures, and N_{Extra} is the number of extra control structures. As there will be more than one model solutions for a given programming exercise,the model solution which has lesser distance D with the student's program will be chosen for the assessment of the student's program. However, this approach of using JUnit framework is limited only to Java programs.

A number of automated assessment systems are evolving. Automated assessment system Mooshak [53], has been evaluated in [72], concluding that the system of feedback needs to be richer.

1.2.1 Automated suggestions for correction of errors: survey of various approaches

LAURA [5] is a system built with a goal to pin-point the semantic errors, or at least localize them; even that is a big help in debugging. In order to establish equivalence of the model program and the student's program this system employs program transformations which are either semantics preserving or are heuristically driven. In case the transformations can not be applied systematically (as opposed to the standardized transformations), the heuristic criteria are used. While trying to establish equivalence, the step where the system finds that it cannot be established, LAURA concludes error and tries to pin-point it or at least localize it. A comparison of automated debuggers as tutoring systems is given in [96], a paper discussing the design of Aadil - a knowledge based automated debugger.

In a recent work Könighofer and Bloem [52] present a novel debugging method for imperative software, featuring both automatic error localization and correction. They use symbolic execution for program analysis. This allows for a wide range of different trade-offs between resource requirements and accuracy of results. Their error localization method rests upon model-based diagnosis and SMT-solving. Error correction is done using a template-based approach which ensures that the computed repairs are readable. Their method can handle all sorts of incorrect expressions, not only under a single-fault assumption but also for multiple faults. A general-purpose SMT-based error finding platform has been reported in [88], a tool for statically checking program assertions and errors. Cordeiro et al. [25] have applied propositional bounded model checking "successfully to verify embedded software but is limited by the increasing propositional formula size and the loss of structure during the translation". They have investigated "the application of different SMT solvers to the verification of embedded software written in ANSI-C." Komuravelli et al. [51] presented an SMT-based symbolic model checking algorithm for safety verification of recursive programs.

In another recent work Rishabh Singh et al. [83] present a new method for automatically grading introductory programming assignments. In order to use this method, instructors provide a reference implementation of the assignment, and an error model consisting of potential corrections to errors that students might make. Using this information, the system automatically derives minimal corrections to student's incorrect solutions, providing them with a quantifiable measure of exactly how incorrect a given solution was, as well as feedback about what they did wrong. They have introduced a simple language for describing error models in terms of correction rules, and formally define a rule-directed translation strategy that reduces the problem of finding minimal corrections in an incorrect program to the problem of synthesizing a correct program from a sketch. Identifying students' misconception of programming is aimed in [41].

1.2.2 Automated assessment of students' programs: survey leading to our formal method based approach

The work in this thesis derives from the equivalence problem of flowchart schema discussed at length in Manna [59]. The equivalence checking problem of FSMDs (EPFSMD) is the same as the equivalence problem of flowchart schema and has found wide application in different areas. Kim et al. [48] have reported an automated formal verification method of the scheduling process using FSMD. In a different work Kim et al. [48] discussed automated formal verification of scheduling with speculative code motions. Matsumoto et al. [60], in their method, at first identify the textual differences between the two programs to find where the equivalence must be checked. Symbolic simulation and validity checking techniques are used to check the equivalence of differences. If the equivalence is not established, their method incrementally extends statements to be verified based on dependency, until the equivalence is proved. For the extensions, the method uses dependence graphs of the programs. Karfa et al. [47] have shown the application of FSMD based method in High Level Synthesis. A given behavioral specification prior to scheduling and its equivalent produced by a scheduler, both of them can be represented by their corresponding FSMDs. In their work they have proposed an algorithm to find the equivalence of two such FSMDs. This FSMD based method has been shown to be equally applicable in several high level code transformations by them. The method has found application in hardware design verification and verification of embedded systems.

1.2.3 Related work for determining equivalence of two expressions

Determining equivalence of two programs basically entails checking whether on giving the same inputs, the two programs produce the same outputs or not. Although this problem is undecidable in general, the problem can be proven to be sound [12, 47, 59] and even complete [56] for some restricted subsets. Since it has to be ascertained that the values output by both the programs are the same, checking equivalence of expressions is at the core of this problem. In this section, we outline the various procedures undertaken to determine equivalence of expressions involving different data-types.

Boolean expressions are the easiest to check for equivalence. Any two Boolean expressions can be converted into either of the two canonical forms namely, conjunctive normal form and disjunctive normal form, and then checked for equivalence based on the fact that whether they have reduced to the same syntactical form or not. However, no such canonical form exists for expressions over integers and consequently, a normal form for such expressions has been proposed in [49] which reduces many of the computationally equivalent expressions into identical syntax, e.g., $0+1\times a\times a+2\times a\times b+1\times b\times b$ is the normal form of $(a+b)^2$. This work has later been adopted in [75], some simplification rules for the normalization grammar have been proposed in [47]; recently, this grammar has been extended to include array references in [14].

The theory of real numbers and bit-vectors, on the other hand, is decidable [20], [85]. Effective solutions for expressions over such variables can be obtained from state-of-the-art SMT solvers. Accordingly, these SMT solvers have been adopted in [13] to handle such datatypes in the context of checking equivalence of programs. It is to be noted that user-defined data-types actually encompass multiple variables of the same or different data-types; consequently checking equivalence of two user-defined variables of the same sort requires application of separate rules for each of its constituent variables. SMT solvers, such as CVC4 [1], provide constructs which can readily capture user-defined data-types as defined in high-level languages, such as C.

1.3 Motivation

Many, if not all, engineering institutions provide mandatory courses on elementary programming for undergraduates. Consequently, the number of students enrolled in such subjects is high. Often, due to lack of sufficient and/or efficient staff, quick and consistent evaluation of students' programs may not be ensured in such places. As a result, automated evaluators which can speedily and consistently assess students' programs have become a necessity for sustenance of the modern academic needs. Owing to the large number of students that academic institutions have to accommodate, automated evaluation of students' programs has received impetus in recent years; an automated evaluator not only ensures speedy evaluation but also consistency in distribution of marks. As an example at IIT Kharagpur a course on programming and data structures is offered to the first year undergraduates, which are approximately 1000 per semester. If 10 assignments each having 10 problems is given to them, a total of 1000x10x10=100000 programs have to be checked per semester, which is a huge task even for a team of 5 instructors and 10 TAs. Another area where automated evaluation is very useful is MOOCs, as discussed in the paper [65] by V. Pieterse. As computer-assisted learning is nowadays becoming popular among students, mainly because it provides an affordable medium for acquiring knowledge and allow remote access; the recent increase in availability of massive open online courses [3] and virtual labs supported by the Indian government [4] bear testimony to this fact. Automated assessment has been observed to have various advantages in such courses [65]. Ala-Mutka [7] mentions speed, availability, consistency and objectivity of assessment, whereas Vujošević-Janičić et al. [89] mention the advantage of feedback available immediately to the students, particularly novices as they can be benefited from early disambiguation of complex ideas. Advantages that are particularly relevant for MOOCs are its potential to facilitate learning and allowing students to practise and get feedback at any time and anywhere Malmi et al. [57], as well as the possibility to give more tasks to the students Enström et al. [30]. Chen [22] claims that use of their system could enable students achieve a higher standard and helped students in meeting such standard.

In this work, we have chosen the FSMD model to represent the programs since equivalence checking of FSMDs is a well studied problem and has found extensive application in translation validation of programs [11, 12, 44, 45, 47, 54]. Specifically, FSMD based equivalence checking is first proposed in [45], which is later developed to handle uniform and non-uniform code motion based optimization techniques in [44, 47, 54]. This method is general enough for checking equivalence of digital circuits as well [46]. A further enhancement of this method can be found in [11, 12] which can additionally handle code motions across loops. Thus, by adopting the FSMD based equivalence checking method into our program evaluation system, we can add to our repertoire a wide range of supported code optimization techniques that may be applied by a student; to the best of our knowledge, no other assessment mechanism has targeted code optimization techniques to such an extent before.

1.3.1 Objective of the thesis

In view of the importance of automated evaluation of students' programming assignments in early programming courses and inadequate research in its design philosophy using equivalence checking and implications, the following research objectives are set for the thesis.

1. To develop a scheme for statement containment analysis of students' programs through equivalence checking.
2. To develop methods to reconcile dissimilarities between FSMDs.
3. To develop supporting techniques for checking and evaluation of students' programs.

As the model used in the thesis for analysis is the finite state machine with data-paths (FSMD). We present the formal description of FSMD model in brief in the following section; a more detailed description of FSMD models is given in paper by Karfa et al. [47]. This description is followed by the equivalence checking mechanism presented by Karfa et al. [47], which we present next with the help of an example for developing the understanding of the method.

1.4 FSMD model

Finite state machine with data-paths (FSMD) is a graph representing a program, having a set of states, known as control states, which are joined through edges. Every edge is associated with conditions over the program variables and some data-transformation. The edges therefore are labeled with conditions and data-transformation pairs.

The FSMD model [33], used in this work to model the golden programs provided by the teacher and the students' programs, is formally defined as an ordered tuple M=$\langle Q, q_0, I, V, O, \tau : Q \times 2^S \rightarrow Q, h : Q \times 2^S \rightarrow U \rangle$, where Q is the finite set of control states, q_0 is the reset (initial) state, I is the set of input variables, V is the set of storage variables, O is the set of output variables, τ is the state transition function, S represents a set of relational expressions involving arithmetic expressions over the members of I and V, U represents a set of assignments of expressions over inputs and storage variables to some storage or output variables and h is the update function capturing the conditional updates of the output and storage variables taking place in the transitions through the members of U.

Below we present an example of program code for computing GCD of two numbers, `z1` and `z2`. The result is stored in `res`. The corresponding FSMD is given in the Figure 1.1 (This example is incorporated from [43]).

```
int z1, z2, res = 1;
scanf(``%d, %d'', &P1, &p2);
z1 = P1;
z2 = P2;
while (z1 != z2){
    if (z1 % 2 == 0)
        if (z2 % 2 == 0)
           res = res * 2;
           z1 = z1 / 2;
           z2 = z2 / 2;
        else
           z1 = z1 / 2;
    else
        if (z2 % 2 == 0)
           z2 = z2 / 2;
        else
           if (z1 > z2)
              z1 = z1 - z2;
           else
              z2 = z2 - z1;
}
res == res * 2;
printf (``%d'', res);
```

Every label of a transition has two parts separated by the symbol '`/`'. On the left of the separator is the condition of execution and to the right is the data-transfer, e,g., if a label is `c/d`, then `c` is the condition of execution (e.g., `(y > 0)`) and `d` is the data transfer (`d` could mean a statement like `x = y + z;`). The data-transfer could be a comma separated list for a path between two adjacent cut-points, as sometimes such labels are used for paths between two neighboring cut-points. Label `-/d` means the

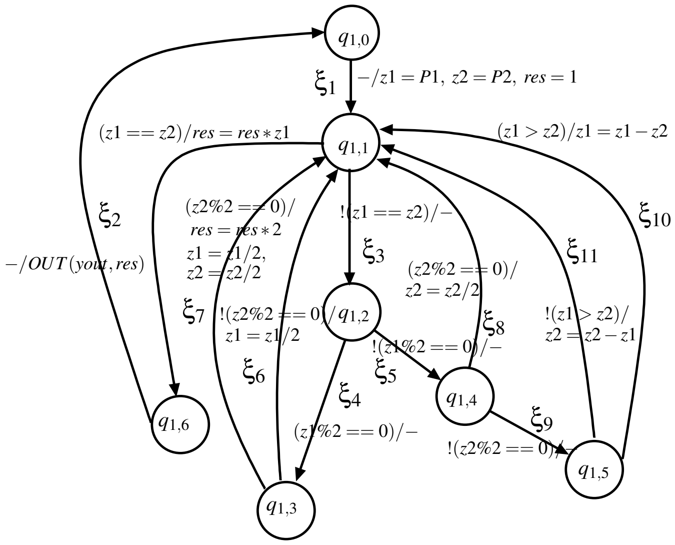


Figure 1.1: M_g: FSMD for golden program for computing GCD.

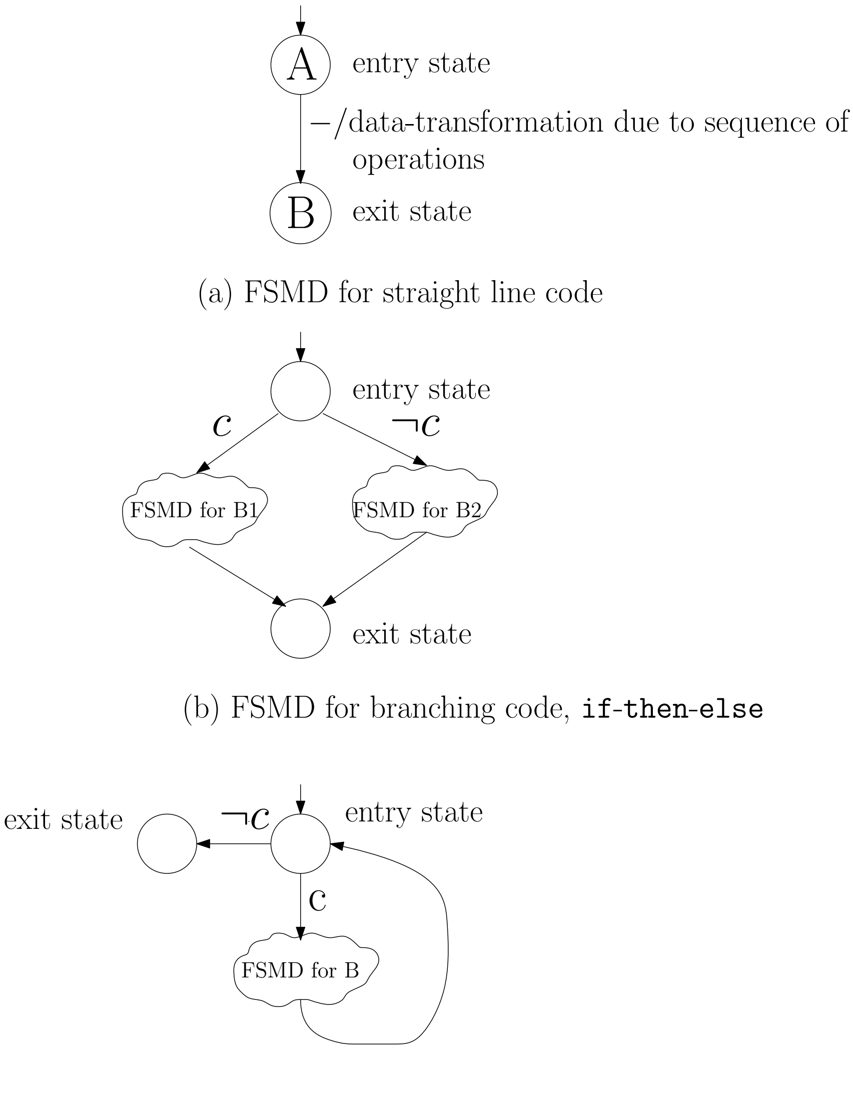
condition of execution is True and the data transformation is d, whereas -/- means the condition of execution is True and there is no data transformation.

In the following discussion we describe the block structured languages, Aho et al. [6], and the FSMD of a block structured program.

Definition 1 (Block structured languages)**.** *A block structured language is a high level programming language, permitting the programs to be made up of blocks, a block may be nested within another block. The blocks are separated by delimiters, which ensure that either the blocks are independent or are nested. A block consists of a sequence of statements and/or other blocks, preceded by declarations of variables.*

Nesting of blocks in a block structured language may go to any depth. Variables declared in the beginning of a block are visible inside the entire block including any nested blocks, unless the same variable is declared inside an inner block. Variable declarations have a nested scope. If a variable is already declared outside and a new declaration of that variable takes place inside an inner block, then the inner declaration has scope throughout the inner block. At the end of the inner block, the outer declaration becomes effective again. Although scoping rules are important, but, in this thesis we are not concerned with the scope rules of variable declarations. In this thesis, the scope of variable declarations are not important, as global variables are used throughout in the thesis. Here, we are more concerned with handling data-transformations along-with their conditions of execution, so we go with the assumption that variables are in general global variables.

In the following we present block structured FSMD in a recursive manner, illustrated in figure 1.2. The base case is the FSMD of a straight line code, which is a sequence of operations, see figure 1.2(a). The FSMD of a branching code, e.g., `if-then-else`, where control is transferred to block B1 for `if` part and to block B2 for `else` part is shown in figure 1.2(b). The FSMD of a loop, e.g., `while` (condition) B is shown in figure 1.2(c), where the control is transferred to block B on the condition *c* being true. In this case, when condition is false, the control is transferred out of the loop.



(a) FSMD for straight line code

(b) FSMD for branching code, `if-then-else`

(c) FSMD for `while` loop code

Figure 1.2: Figure explaining recursive definition of block structured FSMD.

The FSMD of a block structured program represents each block separately, starting with a distinct state and ending in a distinct state. As the FSMD representation is more elegant, a compact FSMD may be obtained by merging the states having a null transition (-/-, condition of execution is True and there is no data transformation) between them or by pushing the not-null transition up or down, as will be evident from the figures 1.3, 1.4, 1.5 and 1.6, where `if` represents the statements of `if`-block and `else` represents the statements of `else`-block. In the thesis, the compacted version of the FSMD has been assumed, wherever the mention is not explicit, without loss of generality.

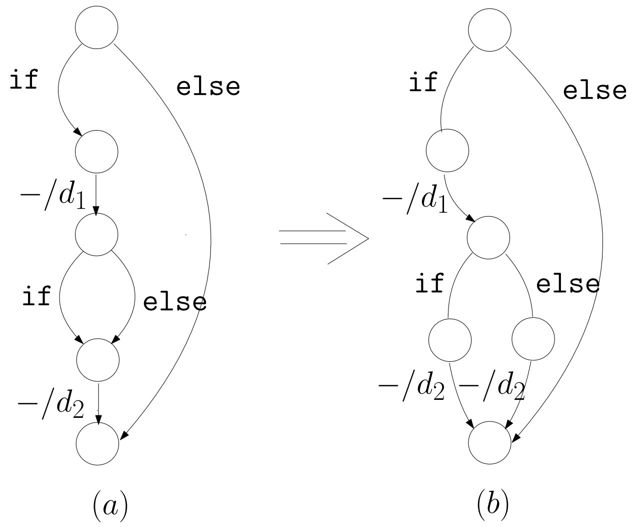


Figure 1.3: Figure explaining compaction of block structured FSMD.

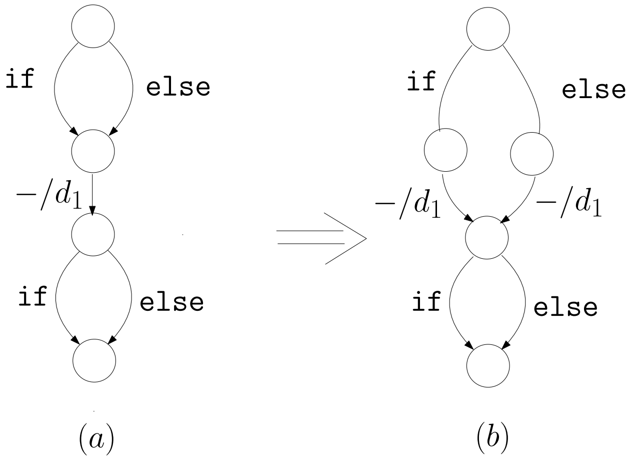


Figure 1.4: Figure explaining compaction of block structured FSMD.

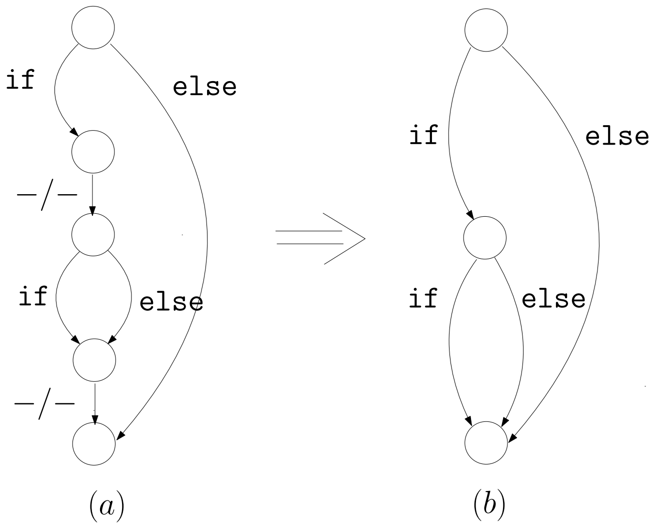


Figure 1.5: Figure explaining compaction of block structured FSMD.

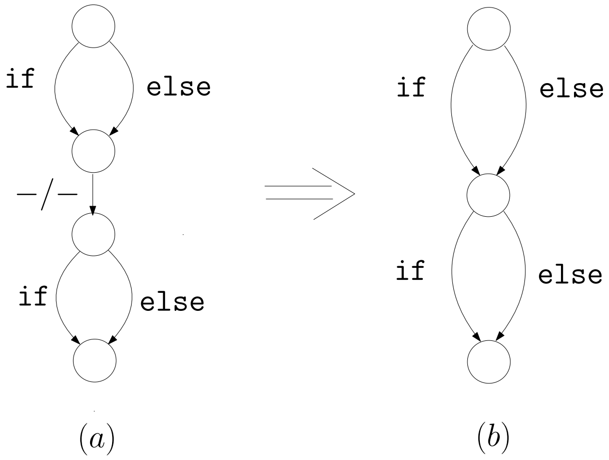


Figure 1.6: Figure explaining compaction of block structured FSMD.

A state transition t is a 4-tuple of the form $\langle q_i, q_f, C_t, r_t \rangle$, where $q_i, q_f \in Q$, $C_t \in 2^S$ and $r_t \in U$, $\tau(q_i, C_t) = q_f$ and $h(q_i, C_t) = r_t$. The third and fourth parameters of t are often represented as $\langle C_t / l_t \leftarrow e_t(\overline{v}) \rangle$. C_t is the condition which is to be satisfied by the values of the variables at the state q_i for the transition to take place, l_t is the left hand side (LHS) variable and $e_t(\overline{v})$ is an arithmetic expression over variables; the value of the variable l_t changes to $e_t(\overline{v})$ when the transition takes place. Depending upon the context, a state transition t is often abbreviated as $\langle q_i, q_f \rangle$, when its initial and the final states are of concern, and also as $\langle C_t / l_t \leftarrow e_t(\overline{v}) \rangle$, when its condition and data transformation are of concern.

A *computation* ν, say, of an FSMD is a sequence of states starting and ending with the reset states without having any occurrence of the reset state in between. The computation ν is associated with two entities, the condition of execution C_ν and its data-transformation r_ν; the condition C_ν is a logical expression over variables in I and captures the condition that should be satisfied by these variables at the initial state (reset state) q_0 for the computation ν to be executed.

The data transformation r_ν is an ordered pair $\langle \mathsf{s}_\nu, \theta_\nu \rangle$, where s_ν represents the symbolic expression values of program variables in V in terms of input variables I at the end of computation ν; the second member θ_ν is a list of symbolic expressions representing the output list produced by the computation ν. [1]

Definition 2 (Equivalence of computations)**.** *The computations ν_1 and ν_2 are equivalent, denoted as $\nu_1 \simeq \nu_2$, if and only if*

1. $C_{\nu_1} \equiv C_{\nu_2}$ *and*

2. $r_{\nu_1} = r_{\nu_2}$.

[1] To determine the equivalence of arithmetic expressions under associative, commutative, distributive transformations, expression simplification, constant folding, etc., we rely on the normalization technique presented in [75] which supports Booleans and integers only and assumes that no overflow or underflow occurs. The method in [87], in contrast, can handle floating point expressions as well since it treats LCM to be a purely syntactical redundancy elimination transformation.

In our context we have two programs: a student's program, whose FSMD is denoted as M_s, and the instructor supplied golden program, whose FSMD is denoted as M_g.

Definition 3 (Containment of FSMD)**.** *An FSMD $M_s = \langle Q_s, q_{s,0}, I, V_s, O, \tau_s : Q_s \times 2^{S_s} \rightarrow Q_s, h : Q_s \times 2^{S_s} \rightarrow U_s \rangle$ is said to be contained in an FSMD $M_g = \langle Q_g, q_{g,0}, I, V_g, O, \tau_g : Q_g \times 2^{S_g} \rightarrow Q_g, h : Q_g \times 2^{S_g} \rightarrow U_g \rangle$, denoted as $M_s \sqsubseteq M_g$, if for the computation ν_s of M_s due to any input on M_s, there exists a computation ν_g of M_g for the same input on M_g, such that $\nu_s \simeq \nu_g$.*

Definition 4 (Equivalence of FSMDs)**.** *The FSMDs M_s and M_g are equivalent, denoted as $M_s \cong M_g$, if*

1. *$M_s \sqsubseteq M_g$ and*
2. *$M_g \sqsubseteq M_s$.*

Due to loops there may be infinite number of computations, hence, containment or equivalence of FSMDs cannot be resolved using the definitions 3 and 4 directly. By introducing cut-points in the loops, a program can be considered to comprise of finite paths, where a path extends from the reset state to a cut-point, a cut-point to another cut-point or from a cut-point to the reset state without having any intermediary cut-points. A path based equivalence checker uses this mechanism to reduce the problem of deciding equivalence of any two computations to the problem of establishing equivalence of the paths which constitutes this computation.

Definition 5 (Cut-point)**.** *A cut point in an FSMD is either the starting or the terminal state, or it is any other state having more than one outgoing transitions.*

The formal notion of paths is introduced below.

A *path* χ in an FSMD model is a finite sequence of states of the form $\langle q_1, q_2, \ldots, q_f \rangle$, where $q_i \in Q, 1 \leq i \leq f$. There are transitions τ_i from q_i to $q_{i+1}, 1 \leq i < f$, q_1 and q_f are cut-points and $q_j, 1 < j < f$, are not cut-points. The initial state q_1 is designated as χ^s and the final state q_f is designated as χ^f. For any $i, 0 \leq i \leq f-1$, the symbol $\chi^{(i)}$ represents the prefix of χ of length *i*; thus, specifically, $\chi^{(i)}$ is the sequence $\langle q_1, q_2, \ldots, q_{1+i} \rangle$.

A path χ is associated with two entities: the condition of execution C_χ and its data-transformation r_χ; the condition C_χ is a logical expression involving variables in the sets V and I, and captures the condition that should be satisfied by these variables at the initial state (reset state) q_0 for the path χ to be traversed.

The data transformation r_χ is an ordered pair $\langle s_\chi, \theta_\chi \rangle$, where s_χ represents the symbolic expression values of program variables in V in terms of input variables I at the end of the computation along the path χ; θ_χ is a list of symbolic expressions representing the output list produced by the computation along the path χ. θ_χ is typically of the form $[OUT(P_{i_1}, e_1), OUT(P_{i_2}, e_2), \ldots]$. More specifically, for every expression e output to port P along the path χ, there is a member $OUT(P, e)$ in the list appearing in the order in which the outputs occur in χ. The condition of execution and the data transformation of a path are computed using the method of symbolic execution.

Definition 6 (Corresponding states and equivalence of paths of two FSMD models)**.**

1. *The reset state of two FSMD models are corresponding states,*
2. *A reset state of one FSMD cannot be a corresponding state of a non-reset state of another FSMD,*
3. *Two paths χ_s of M_s and ξ_g of M_g are said to be equivalent if, χ_s^s and ξ_g^s are corresponding states, $C_{\chi_s} \equiv C_{\xi_g}$ and $r_{\chi_s} = r_{\xi_g}$,*
4. *If two paths χ_s and ξ_g are equivalent, then their final states χ_s^f and ξ_g^f are corresponding states.*

In the FSMDs M_s and M_g, the states are represented as $q_{s,i}$ and $q_{g,j}$ respectively. It is to be noted that two equivalent programs must have the same set of output variables. When we look at the output value of a variable, the output value of the counterpart variable in other FSMD must also be the same. Thus, equivalence of θ_χ depends on the equivalence of s_χ. In the remaining part of this chapter, therefore, computation of s_χ is described; for reason of brevity, the computation θ_χ is omitted.

Definition 7 (Path cover of an FSMD)**.** *A path cover of an FSMD M is a finite set of paths $P = \{\chi_1, \chi_2, \chi_3, \ldots, \chi_k\}$, such that any computation ν of M can be viewed as a concatenation of some paths $\in P$.*

A path cover is thus a finite set of all paths from one cut -point to the next cut-point, in which there is no cut-point in between [32]. Identification of equivalent paths from the two FSMDs and the corresponding states of the two FSMDs go hand in hand as indicated in definition 6.

The above discussion leads to the following theorem [10, 43], for which an inductive proof is given below.

Theorem 1 (FSMD containment)**.** *Given two FSMDs M_s and M_g, let there be a finite path cover $P_s = \{\chi_1, \chi_2, \ldots, \chi_{l_1}\}$ of M_s and a finite set of paths $P_g = \{\xi_1, \xi_2, \ldots, \xi_{l_2}\}$ of M_g such that for all corresponding state pairs, $\langle q_{s,i} \in M_s, q_{g,j} \in M_g \rangle$, for any path $\chi_m \in P_s$ starting from $q_{s,i}$, there exists a path $\xi_n \in P_g$ starting from $q_{g,j}$ such that $\chi_m \simeq \xi_n$, then M_s is said to be contained in M_g ($M_s \sqsubseteq M_g$).*

Proof: P_s being a path cover from M_s, every computation ν_s from M_s is along a concatenated path $[\chi_{i_1}\chi_{i_2}\ldots\chi_{i_r}]$ from P_s, originating at the reset state $q_{s,0}$ and ending also on the same reset state. We need to prove that there exists a computation ν_g in M_g so that $\nu_s \simeq \nu_g$. Let ν_g be the concatenated path $[\xi_{j_1}\xi_{j_2}\ldots\xi_{j_l}]$, where $\chi_{i_k} \simeq \xi_{j_k}, 1 \leq k \leq l$, as per the hypothesis in the theorem. In the following we prove that ν_g is a computation of M_g and $\nu_s \simeq \nu_g$.

Let $\nu_{s,l}(\nu_{g,l}), 1 \leq k \leq r$, be a subsequence of $\nu_s(\nu_g)$. We prove by induction on l that for any valuation σ of the varaiables of M_s (and M_g), which results in the computation ν_s in M_s, the computation ν_g would result in M_g and $\nu_{s,l} \simeq \nu_{g,l}$. Note that M_s and M_g are both deterministic, hence for each valuation σ, there is a unique computation in M_s.

Basis:($l = 1$): $\nu_{s,1} = [\chi_{i_1}]$; since ν_s is a computation of M_s, the initial state $\chi^s_{i_1}$ of the path χ_{i_1} is the reset state of M_s. Since reset state of M_s can have correspondence with only the reset state of M_g, the path $\xi^s_{j_1}$ must be the reset state. By construction of ν_g, $\chi_{i_1} \simeq \xi_{j_1}$; hence $\nu_{s,1} \simeq \nu_{g,1}$. Since M_s and M_g are deterministic, χ_{i_1} is the only path of M_s and ξ_{j_1} is the only path of M_g which bear equivalence under σ. Also, the final states $\chi^f_{i_1}$ and $\xi^f_{j_1}$ are corresponding states as per the clause 4 of Definition 6.

Induction Hypothesis: Let $\nu_{s,l} = [\chi_{i_1}, \ldots, \chi_{i_l}] \simeq \nu_{g,l} = [\xi_{j_1}, \ldots, \xi_{j_l}]$ and the final states $\chi^f_{i_l}$ and $\xi^f_{j_l}$ are corresponding states. Thus, the respective transformed values of σ along the subsequences $\nu_{s,l}$ and $\nu_{g,l}$, namely, $r_{s,l}(\sigma)$ and $r_{g,l}(\sigma)$ are equal, i.e.,

$r_{s,l}(\sigma) = r_{g,l}(\sigma)$.

Inductive Step: R.T.P. $\nu_{s,l+1} = [\chi_{i_1}, \ldots, \chi_{i_{l+1}}] \simeq \nu_{g,l+1} = [\xi_{j_1}, \ldots, \xi_{j_{l+1}}]$.

From induction hypothesis, for the valuation σ, the transformed value at $\chi^s_{i_{l+1}} = r_{s,l}(\sigma)$; from the fact that M_s is deterministic, there is only one path $\chi_{i_{l+1}}$ possible under $r_{s,l}(\sigma)$ and $C_{s,l+1}(r_{s,l}(\sigma))$ is true. Since $\chi_{i_{l+1}} \equiv \xi_{j_{l+1}}$, $C_{g,l+1}(r_{g,l}(\sigma))(\equiv C_{s,l+1}(r_{s,l}(\sigma)))$ must be true. Hence, by the fact that M_g is deterministic, $\xi_{j_{l+1}}$ is the only possible path in M_g for the valuation $r_{g,l}(\sigma)$. Also, $\chi^f_{i_{l+1}}, \xi^f_{j_{l+1}}$ are corresponding states and $r_{s,l+1}(\sigma) = r_{g,l+1}(\sigma)$. Hence, the inductive proof is complete. Now, specifically, for $l = r$, $\nu_{g,l} = \nu_g \simeq \nu_{s,l} = \nu_s$ and ν_g is the only sequence of paths followed by M_g under σ; also ν^f_g has correspondence with ν^f_s and the later being the reset state of M_s, the former is the reset state of M_g. So ν_g is a computation of M_g. ■

We may note that *i)* there is no unique selection of cut-points and *ii)* it cannot be guaranteed that for a given path cover of an FSMD, from a particular way of selecting cut-points, there will be a corresponding set of paths that are equivalent in the other FSMD. The example below shows how an initial selection of cut-points may be changed by equivalence checking method based on path extension [44, 47, 54], which uses Theorem 1 for finding the equivalence of FSMDs.

Example 1. The golden FSMD $M_{g_{GCD}}$ is shown in Figure 1.7(a) and Figure 1.7(b) shows the FSMD $M_{s_{GCD}}$, from the student's program. The FSMDs $M_{g_{GCD}}$ and $M_{s_{GCD}}$ are from programs for computing the greatest common divisor of two integers. The states $\{q_{g,0}, q_{g,1}, q_{g,2}, q_{g,3}, q_{g,4}, q_{g,5}\}$ for $M_{g_{GCD}}$ and the states $\{q_{s,0}, q_{s,1}\}$ for $M_{s_{GCD}}$ are chosen cut-points initially. The notation $q_m \twoheadrightarrow q_n$ is used to denote a path starting at the state q_m and going to the state q_n. The notation $q_m \xrightarrow{cond} q_n$, which denotes the path from q_m to q_n corresponding to the condition *cond* being satisfied, is used in order to differentiate between the paths which originate from the same state due to different conditions. It is to be noticed that if a mismatch is found between two paths being compared, the path that is extended is the one associated with the condition of execution that is weaker. In the following we give the sequence of execution of the algorithm given in [47].

1. ξ_1 is found as the equivalent path for χ_1;

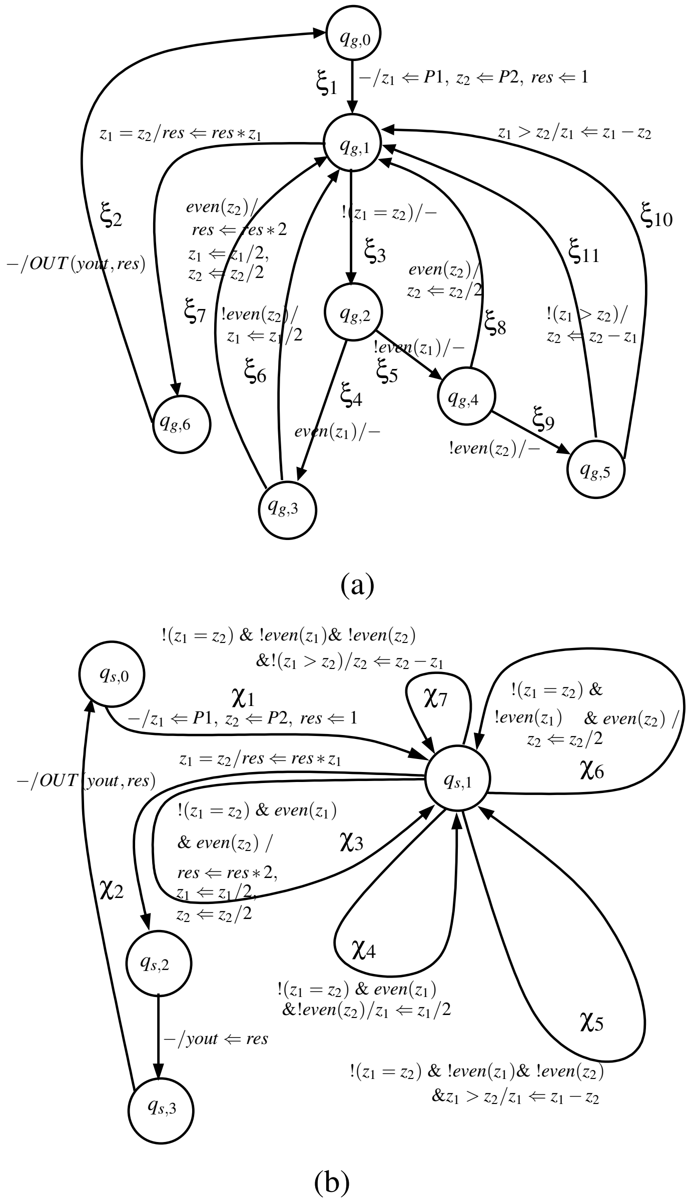


Figure 1.7: (a) M_g: Golden FSMD. (b) M_s: Student's FSMD.

2. ξ_2 is found as the equivalent path for χ_2;

3. fails in order to find the equivalent path for ξ_3, therefore, extension is done; the paths extended are $\xi_3\xi_4$ and $\xi_3\xi_5$;

4. fails in order to find the equivalent path for $\xi_3\xi_4$, therefore, extension is done; $\xi_3\xi_4\xi_6$ and $\xi_3\xi_4\xi_7$ are the extended paths;

5. fails in order to find the equivalent path for $\xi_3\xi_5$, therefore, extension is done; $\xi_3\xi_5\xi_8$ and $\xi_3\xi_5\xi_9$ are the extended paths;

6. χ_4 and χ_3 are found the equivalent paths of $\xi_3\xi_4\xi_6$ and $\xi_3\xi_4\xi_7$ respectively;

7. χ_6 is found to be the path equivalent to $\xi_3\xi_5\xi_8$;

8. fails in order to find the path equivalent to $\xi_3\xi_5\xi_9$, therefore, extension is done; $\xi_3\xi_5\xi_9\xi_{10}$ and $\xi_3\xi_5\xi_9\xi_{11}$ are the paths after extension;

9. χ_5 and χ_7 are found the equivalent paths of $\xi_3\xi_5\xi_9\xi_{10}$ and $\xi_3\xi_5\xi_9$ ξ_{11} respectively.

It is thus found that the corresponding states in this example is given by the set $\{\langle q_{g,0}, q_{s,0}\rangle, \langle q_{g,1}, q_{s,1}\rangle\}$. It is obtained from Theorem 1 that $M_g \sqsubseteq M_s$ and vice versa, i.e., $M_s \sqsubseteq M_g$ again from Theorem 1. Hence, it is concluded that M_g and M_s are equivalent from Definition 4. The path covers of the golden and the student's FSMDs, P_g and P_s are given below. There is a one-to-one correspondence between the members in the two path covers, in terms of path equivalence.

$$
\begin{aligned}
P_g = \{\ & q_{g,0} \twoheadrightarrow q_{g,1}, \\
& q_{g,1} \xrightarrow{z_1=z_2} q_{g,0}, \\
& q_{g,1} \xrightarrow{!(z_1=z_2)} q_{g,2} \xrightarrow{even(z_1)} q_{g,3} \xrightarrow{even(z_2)} q_{g,1}, \\
& q_{g,1} \xrightarrow{!(z_1=z_2)} q_{g,2} \xrightarrow{even(z_1)} q_{g,3} \xrightarrow{!even(z_2)} q_{g,1}, \\
& q_{g,1} \xrightarrow{!(z_1=z_2)} q_{g,2} \xrightarrow{!even(z_1)} q_{g,4} \xrightarrow{even(z_2)} q_{g,1}, \\
& q_{g,1} \xrightarrow{!(z_1=z_2)} q_{g,2} \xrightarrow{!even(z_1)} q_{g,4} \xrightarrow{!even(z_2)} q_{g,5} \xrightarrow{z_1>z_2} q_{g,1}, \\
& q_{g,1} \xrightarrow{!(z_1=z_2)} q_{g,2} \xrightarrow{!even(z_1)} q_{g,4} \xrightarrow{!even(z_2)} q_{g,5} \xrightarrow{!(z_1>z_2)} q_{g,1}\ \}, \\
P_s = \{\ & q_{s,0} \twoheadrightarrow q_{s,1}, \\
& q_{s,1} \xrightarrow{z_1=z_2} q_{s,0}, \\
& q_{s,1} \xrightarrow{!(z_1=z_2)\&even(z_1)\&even(z_2)} q_{s,1}, \\
& q_{s,1} \xrightarrow{!(z_1=z_2)\&even(z_1)\&!even(z_2)} q_{s,1}, \\
& q_{s,1} \xrightarrow{!(z_1=z_2)\&!even(z_1)\&even(z_2)} q_{s,1}, \\
& q_{s,1} \xrightarrow{!(z_1=z_2)\&!even(z_1)\&!even(z_2)\&z_1>z_2} q_{s,1}, \\
& q_{s,1} \xrightarrow{!(z_1=z_2)\&!even(z_1)\&!even(z_2)\&!(z_1>z_2)} q_{s,1}\ \}.
\end{aligned}
$$

■

It is seen in the example above that the approaches using path extension [44, 47, 54] modify the path cover in the initial set. The property characterized in the Theorem 1 is satisfied by the resultant path covers. In the subsequent chapter, a method is devised whereby a path based approach is used to find out such containment. A path cover is obtained in this work by treating the start and final states and the branching states (i.e., states having number of outgoing transitions > 1) of the FSMD to be cut-points, and by taking the path from one cut point to the next.

The FSMDs M_s and M_g are checked for equivalence using the FSMD equivalence checker, which obtains the paths of the two FSMDs and starts to check equivalence of paths originating from the initial states of M_s and M_g, which are assumed to be the corresponding states. In general, the equivalence checker tries to identify for any path χ_s of M_s, an equivalent path ξ_g of M_g originating from the state that corresponds to χ_s^s.

If it fails, then it extends one of these paths, p say, tries to obtain the equivalent of the extended path with the original path of the other FSMD. Specifically, let $C_{\xi_g} \rightarrow C_{\chi_s}$, in which case χ_s will be extended by concatenating a path emanating from χ_s^f. This concatenated path of M_s is compared for equivalence with ξ_g. This extension may go on for several steps. However, an extension never moves through a loop. After several extensions, the equivalence checker may find an equivalent path, or meet some loop cut-points, or the final state of the FSMD through which extension is no longer possible. In case of extensions, after all the paths starting from the corresponding state of ξ_g^s are found to fail to have equivalence with ξ_g, the equivalence checker terminates, indicating that the path ξ_g does not have an equivalent path in χ_s. (In case $C_{\chi_s} \rightarrow C_{\xi_g}$, then path ξ_g is extended).

The failure of equivalence checker indicates the student's program is at fault and thus there is further need to analyze exactly what is the fault. A statement containment checking mechanism has been devised in order to ascertain the type of fault the student's program has. Using statement containment checking the fault in the student's program can be categorized into one of the broad types of errors, which are then dealt with by the corresponding error handling routine. The modification of equivalence checking mechanism by adding statement containment checking and development of error handling mechanisms is one of the primary objectives of the thesis. The statement containment analysis and other objectives of this thesis are described in the following section.

1.5 Normalization

The equivalence checking of two paths χ of FSMD M_s and ξ of FSMD M_g involves matching syntactically the C_χ with C_ξ and the r_χ with r_ξ.

In doing so, a problem arises because the student may write the same expression in a different way. For example, the expression `(x + y) * w + v` in the golden program, may be written as `x * w + y * w +v` in the student's program. The problem here is, how to compare the two equivalent expressions as above, unless they are syntactically identical. This brings us to the question of canonical form of the expressions. By having canonical form in an arithmetic, we can say that all equivalent expressions will have the identical canonical form, for example in switching algebra the sum of products or the product of sums are examples of canonical forms. Canonical form exists in switching algebra, as the same is decidable. However, in the integer arithmetic such is not the case, as there is no canonical form possible. The best we can do is to evolve a normal form in integer arithmetic, as no canonical form exists for integer expressions. Canonical form is a special form of normal form, it renders any computationally equivalent formulae to a single form. Normalization is way to put the expressions in a fixed syntactic structure, called the normal form, whereby the chances of two equivalent formulae becoming syntactically identical increases, but still there is a possibility of some equivalent formulae not being syntactically identical. This follows from the fact that $\langle \mathbb{N}, +, \times \rangle$ is incomplete, as per Gödel's Incompleteness theorem [84], and the set $\mathbb{Z}$ of integers is isomorphic to $\mathbb{N}$, the set of non negative numbers. This means we cannot have an axiomatic system, in which there will be proof of all true statements. The undecidability of integer arithmetic is a consequence of incompleteness, but not vice-versa. This undecidability implies that there is no canonical expression possible for integer arithmetic expressions. We thus have to be satisfied with only some normal forms, which only increase the the possibility of two equivalent expressions becoming syntactically identical. The idea for normalization was originally given in detail in the paper by Sarkar and De Sarkar [75] and has been used in [10, 43]. It may, however, be noted that all equivalent expressions do not become syntactically identical, when they are normalized, due to the reasons explained above.

The grammar for normalization, which includes uninterpreted function constants/primaries, is given as follows:

1. $S \rightarrow S + T \big| c_s$, where c_s is an integer.
2. $T \rightarrow T * P \big| c_t$, where c_t is an integer.
3. $P \rightarrow \mathrm{abs}(S) \big| (S) \bmod (C_d) \big| f(listS) \big| S \div C_d \big| v \big| c_m$, where v is a variable, and c_m is an integer.
4. $C_d \rightarrow S \div C_d \big| (S) \bmod (C_d) \big| S$,
5. $listS \rightarrow listS, S \big| S$.

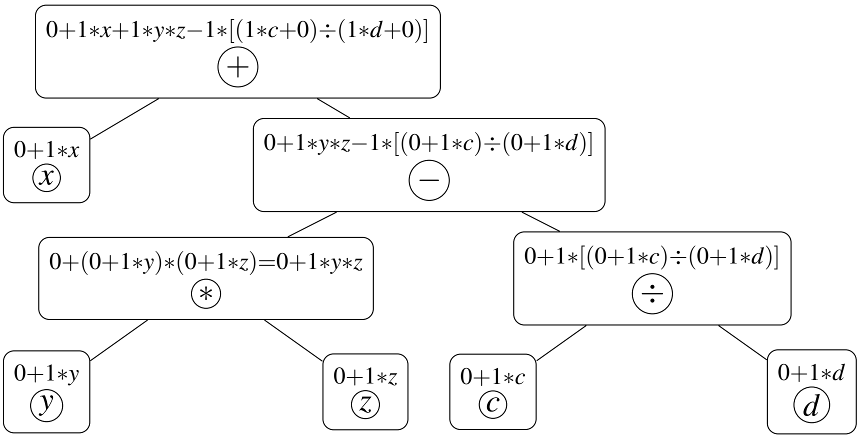

0+1∗x+1∗y∗z−1∗[(1∗c+0)÷(1∗d+0)]
+
0+1∗x
x
0+1∗y∗z−1∗[(0+1∗c)÷(0+1∗d)]
−
0+(0+1∗y)∗(0+1∗z)=0+1∗y∗z
∗
0+1∗[(0+1∗c)÷(0+1∗d)]
÷
0+1∗y
y
0+1∗z
z
0+1∗c
c
0+1∗d
d

Figure 1.8: Expression tree for method of obtaining normalization for $x+y*z-(c\div d)$.

In the above grammar, the nonterminals S, T, P stand for (normalized) sums, terms and primaries, respectively, and C_d is a divisor primary. The terminals are the variables belonging to the set of input and storage variables, the interpreted function constants abs, mod and $\div$ and the user defined uninterpreted function constants f. An example of user defined uninterpreted function constant is $f(v_1,v_2,4)$, which will be normalized as $f(0+1*v_1,0+1*v_2,4)$. In addition to the syntactic structure, all expressions are ordered as follows: any normalized sum is arranged by lexicographic ordering of its constituent subexpressions from the bottom-most level, i.e., from the level of simple primaries assuming an ordering over the set of variables; among the function terminals, abs $\prec \div \prec$ mod $\prec$ uninterpreted function constants, where the symbol $\prec$ stands for the ordering relation "precedes". As such, all function primaries, including those involving the uninterpreted ones, are ordered in a term in an ascending order of their arities.

The equivalence checking of two paths χ of FSMD M_s and ξ of FSMD M_g thus involves matching syntactically the normalized form of C_χ with that of C_ξ and the normalized form of r_χ with that of r_ξ.

As an example of normal forms, the method of obtaining normal form for the expression $x+y*z-(c\div d)$ is shown in Figure 1.8 with the help of an expression tree for the normalized form.

The leaves of the tree in Figure 1.8 show the normal forms of the single variable expressions x, y, z, c and d, which are $0+1*x, 0+1*y, 0+1*z, 0+1*c$ and $0+1*d$, respectively. The intermediate nodes show the resultant normal form after the operation at the node has been performed. For instance, the normal form of $y*z$ is $0+(0+1*y)*(0+1*z)$, which evaluates to the form $0+1*y*z$. The root of the tree shows the normal form of the overall expression $x+y*z-(c \div d)$, which happens to be $0+1*x+1*y*z-1*[(0+1*c)(0+1*d)]$.

Normalization is a well defined procedure. Normalization is a sound but not complete mechanism for checking equivalence of arithmetic expressions. The normalization scheme used here is adapted from [43].

1.6 Contributions of the thesis

The thesis attempts to achieve its objectives by addressing specific research issues. The following sub-sections describe the research objectives, the motivation and the work done to address the research objectives. The results, and discussion on the results have been provided in the respective chapters.

1.6.1 A scheme for statement containment analysis of students' programs through equivalence checking

In this part of the work the research objectives were identified as *i)* developing a scheme for containment analysis of students' programs through equivalence checking, *ii)* classifying the errors in programs into broad categories and *iii)* devising strategies for error diagnosis for each class of errors in the programs. Work done for this part is described below.

1. The existing path extension based FSMD equivalence checking approach, which compares two given FSMDs having same variable naming and does a depth first analysis for equivalence of paths from one cut point to the next in the two FSMDs, was modified to incorporate a statement containment checking approach.

In statement containment checking, it is examined whether the assignment statements in a path are contained in the other FSMD or not. We define the statement containment formally in the following discussion. *For brevity we shall refer statement containment checking as containment checking henceforth throughout in this thesis.*

2. Using the above approach of containment checking of the FSMDs based on comparison between cut-point to cut-point path pairs in them, the containment could be divided into following broad categories discussed at length in section 2.2.2 , viz., *i)* unordered path-wise both way contained, *ii)* path-wise one way contained, and *iii)* path-wise un-contained. A combination of the categories of unordered path-wise both way contained and path-wise one way contained, known as unordered path-wise both way contained and path-wise one way contained for faulty branching, is also worth noting as this case is reported in the error of parenthesis skipping, mentioned below. This case is therefore treated as a seaparate category of containment in addition to the three above. These categories of containment can be mapped to various types of errors in the programs with respect to the golden model, which may creep in while programming in detail in section 2.3 viz., *i)* dependency violation, *ii)* parenthesis skipping of various types and *iii)* error of missing code segment. The details of these errors and their associated type of containment have been shown in table 2.1.

3. Algorithms have been developed for reporting the type of error in the student's program and suggesting the correct portion of the code from golden program, corresponding to the erroneous code of student's program.

Containment checking and analysis is a novel idea and a major contribution in this thesis. It has not been attempted before to the best of our knowledge.

1.6.2 Methods to reconcile dissimilarities between FSMDs

In this part of the work preprocessing requirements of programs have been aimed at. The research objectives for this part of the work were identified as *i)* developing methods to support automated evaluation, in cases where programs have conditional constructs, which should obey precedences and *ii)* to develop a scheme for variable mapping in programs. Work done for this part is described below.

The first aspect mentioned above is due to the fact that logic in programs demands that in a nesting of if statements there are conditions that have to be evaluated in a certain order, but the students may violate that order. The student's program, therefore, has to be subjected to a preprocessing step, before equivalence checking is done. The objective of preprocessing is that the nested conditions should conform to some rule of precedence and that a mapping of the names of variables has to be evolved.

In the following part we first describe the variable mapping problem. Since the students will be using variable names different from those in the golden program, we will have to evolve a mapping or an association of the variables used in the student's program, with the ones used in the golden model. An algorithm has been developed for variable mapping between two programs. This is done as the FSMD based equivalence checking assumes the variable names to be the same in the two programs under examination, without which the equivalence checking is not possible. The variable mapping algorithm is an FSMD driven algorithm in the sense that it prepares the FSMD models of both the programs, compares their paths for similarity of conditions and data-transformation in a depth first manner and tries to establish a mapping between the variables, which assume equivalent symbolic values after traversing a path from a cut point to the next.

1.6.3 Supporting techniques for checking and evaluation of students' programs

The research objectives in this part of the work were to develop supporting techniques for checking and evaluation of students' programs such as *i)* checking equivalence of approximately equivalent expressions and *ii)* develop a marking scheme for the programming exercises.

To address the above research questions, in this work additional supporting techniques for handling the student programs have been focused. A description of this work is given below.

Checking equivalence of approximately equivalent expressions

Comparison of approximately equivalent functions [78, 79] may be required as an aid to equivalence checking discussed earlier. This may be because equivalence checking is not able to determine such equivalences, where two expressions which are approximately equivalent are to be examined for equivalence. An example of two equivalent expressions could be $sin2\theta$ and $2sin\theta cos\theta$. The equivalence checker cannot find out that these two expressions are equivalent. In this work, therefore, we have used a randomised simulation based approach in which a function is examined at random points in the domain of the other function. The closeness in their values at most of the points may indicate their approximate equivalence. A randomised decision procedure has been presented to establish the equivalence and the results obtained have established the suitability of the method.

Marking scheme for the programming exercises

A marking scheme for the programming exercises has been developed and tested to perform satisfactorily. Based on FSMD equivalence checking, using propagation vectors, we have also suggested an improved method [77]. This method also suggests the as soon as possible (ASAP) and as late as possible (ALAP) marking strategies; they are based on the observation that a student program should not be penalized for some apparent error as it may have been taken care of at a later stage in the program.

ASAP stands for *as soon as possible* and ALAP stands for *as late as possible*. The ASAP and ALAP strategies are based on the observation that a student may have apparently missed out some code in the initial part of his program, which he may have included at a later stage in the program. Hence, a student program should not be penalized for some apparent error in the beginning, which might have been taken care of in the later part of the program. The ALAP and ASAP passes of evaluation scheme are explained in the section " 4.2.4 A value propagation based automated program evaluation scheme."

An algebraic formula has been suggested to compute marks for the student's program, using some constants, which need to be empirically established. While evaluating the student program (FSMD), we keep track of the number of mismatched variables, N_v say, and the number of mismatched paths, N_p say. The marks M given to a student out of the full marks F is calculated based on the formula. In addition, the marks obtained can be increased or decreased by another factor which we term as *leniency*, since from our experience we have found that teachers tend to award marks more generously during semester examinations than class tests.

In the block diagram of figure 1.9, the sequence of invoking various modules has been shown. The student's program is checked for violation of order of precedences of conditional construct such as if- else if. The program not meeting the precedence is then informed to the student, who can submit his program later, after correcting the precedences. The student's program having the correct precedences and the golden program are then subjected for C to FSMD conversion, thereby generating the respective FSMDs. The two FSMDs thus obtained cannot be subjected to equivalence checking just at this stage, because the student's program may use a different set of variable names than the golden program. The variables in the FSMD of student's program have therefore to be renamed first and then we can have the equivalence checked. The current implementation supports the variable renaming as a separate module, however, it can be modified to do variable renaming and equivalence checking hand in hand, as it proceeds with the equivalence checking, starting from the start states of both the FSMDs. In the current implementation, however, equivalence checking is to be followed by variable renaming. In equivalence checking, if it is found that the two FSMDs are equivalent, then the student's program is correct, however if such is not the case then the containment analysis is done to find out how much of the correct code has been written by the student. As a result of first time containment checking, the student's FSMDs is declared to have one of the four types of containment as compared with the golden program, thus indicating some error, which is informed to the student for correction. After first correction, the student may submit again for subsequent equivalence checking, and to find out the next error. This loop may be continued till the student's program is free from all errors. The loop may be mechanized in the future work. The current implementation assumes the student to be in the loop, doing the correction and resubmitting, if needed and hence such a loop is not shown in the block diagram. The evaluation then can be done on the basis on the number of times the errors were have to be corrected. Presently we rely on a variable mapping and path equivalence based evaluation of the programs, which has been found to give satisfactory results, as discussed in chapter 4.

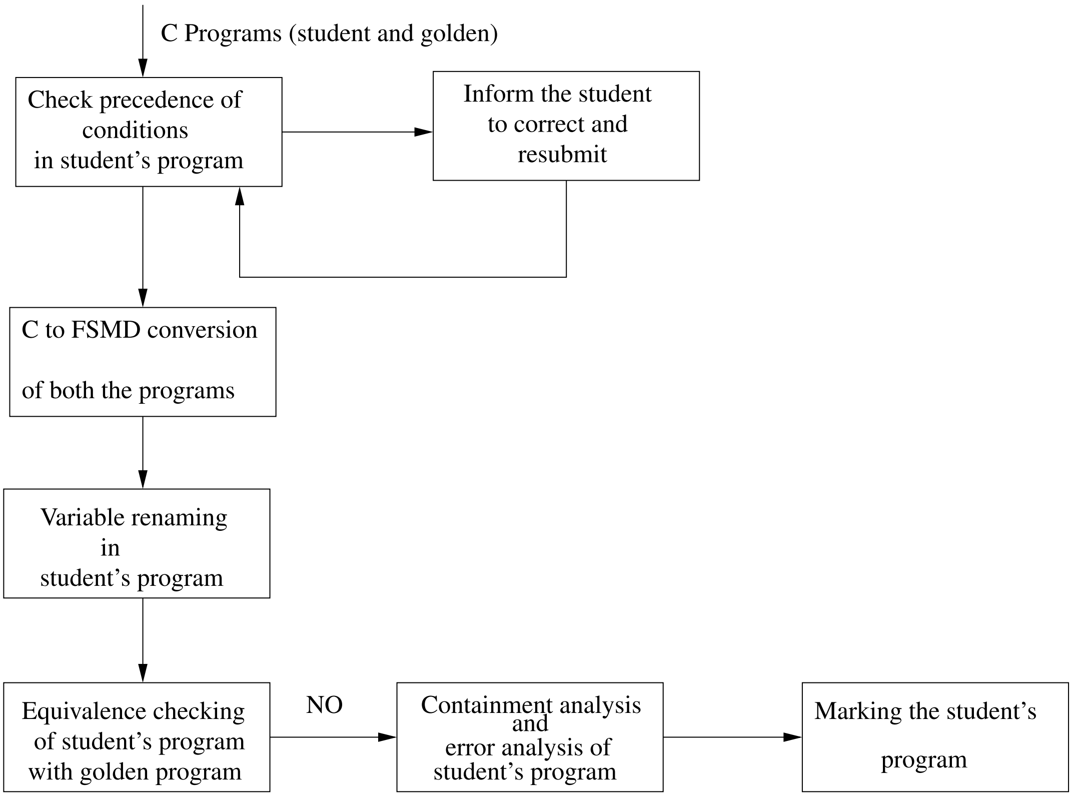

C Programs (student and golden)
Check precedence of conditions in student's program
Inform the student to correct and resubmit
C to FSMD conversion of both the programs
Variable renaming in student's program
Equivalence checking of student's program with golden program
NO
Containment analysis and error analysis of student's program
Marking the student's program

Figure 1.9: Block diagram for automated assessment

1.7 Thesis organization

The organization of the thesis, based on our research work, is as follows.

Chapter 1 Introduction

This chapter discusses state of the art, the need, motivation and objectives of our research work.

Chapter 2 Containment analysis of students' programs through equivalence checking

This chapter provides discussion on a modified equivalence checking algorithm, by introducing the notion of containment checking. Using the containment checking algorithm the path-containment of FSMD of a student's program can be categorized into one of the three broad categories. These categories of containment, mentioned in the previous section 1.6, can be mapped to various types of errors in the programs with respect to the golden model, which may creep in while programming. The details of these errors and the methods for detecting and reporting them are described in this chapter.

Chapter 3 *Methods to reconcile dissimilarities between FSMDs arising from students' programs*

This chapter includes at first the discussion on constructs like if-else-if, which require the conditions to follow a precedence order. If there is a violation of precedence in occurrence of conditions, then the program can be immediately declared to be erroneous. This chapter further has a discussion on variable mapping problem. This problem is important as the equivalence checking method requires that the names of the variables used must be the same, in the programs whose equivalence is to be established. The programs written by students, however, cannot be expected to have the same naming of variables as in the instructor's program. The issue of mapping the variables in two programs is addressed in detail in this chapter.

Chapter 4 *Supporting techniques for checking and evaluation of students' programs*

The first topic covered in this chapter is the automated checking of equivalence of expressions, as a student may write in his program an expression, which is approximately equivalent to the one in the instructor-supplied program. The expression checking method is a randomised simulation based method, capable of establishing whether two given expressions are approximately equivalent in the given range of interest, with a high probability or confidence. The next topic in this chapter is an automated evaluation scheme for student programs.

Chapter 5 *Conclusion and scope for future work*

This chapter concludes the thesis and indicates the scope of future work in the area of automated assessment.

1.8 Assumptions in the thesis

The assumptions in the thesis are as follows:

1. The programs have no structures, pointers, arrays, function calls, classes. Though, in principle, the fields of a structure can be enumerated as distinct variables.

2. The programs may have loop constructs and branching constructs like `for, do-while, if-else` etc. Switch-case construct is not considered.

3. The student and golden programs must have same variable names when subjected to equivalence checking. (This is assumed in entire Chapter 2).

4. More than one golden programs are available with different approaches.

5. The programming language used is C, but no particular language is assumed. However, a block structured language 1 is assumed, so that the FSMD obtained is structured.

6. Use of block structured language 1 lets us assume goto less programming.

7. All variables are global.

8. The programs are syntactically correct.

9. Programs errors include the errors like dependency violation, skipping parenthesis, missing code, precedence in constructs like `if-else if` and approximate equivalence of mathematical expressions.

10. Advanced data structures are not considered.

11. Only programs which can be subjected to basic equivalence checking method are considered.

1.9 Conclusion

In this chapter we have introduced the work done in the thesis. Our literature survey has motivated our work towards automatic evaluation of students' programs using FSMD equivalence checking, as FSMDs can conveniently be used as program representation model. The programs of the students are compared with the golden models provided by the instructor by obtaining their respective FSMDs, first by the equivalence checking method. For this reason, we introduced the FSMD equivalence checking and the related terminology and presented an example of the working of path-extension based equivalence checking. A brief overview of the various topics undertaken in the thesis was subsequently presented. The organization of this thesis was given at the end of the chapter.

Path-extension based equivalence checking is a powerful method to determine equivalence of two FSMDs. Once the equivalence checker fails to find the equivalence between the student's faulty program and the golden program, the reason for non-equivalence is to be detected automatically. A subsequent run of containment checker is carried out to find the containment of paths of the student's FSMD in the FSMD of golden model; error type can be diagnosed based on the report of containment checking. The next chapter is all about the containment checking mechanism.

Chapter 2

Containment analysis of students' programs through equivalence checking

2.1 Introduction

In the previous chapter, an introduction to equivalence checking was provided, which forms the basis of our work in this chapter and the most of the thesis. This chapter aims at extending this method for checking the correctness of a student program and also tries to implement an automated correction system for some cases of faulty programs. Actually, the method of equivalence checking of two given FSMDs involves some code transformation in the form of normalization of the computations as given in paper by Sarkar and De Sarkar [75], in order to achieve some consistency in their representation. The FSMDs with the normalized computations are then compared for establishing the equivalence. Cut-points are introduced in the FSMDs for equivalence checking. For every path between a cut-point to the next cut-point in one FSMD, the equivalence checker looks for an equivalent path in the other FSMD. This process is then repeated with the FSMDs interchanged, in case one way equivalence is established. Once this method reports that both FSMD are different, then our method further checks the containment of the reported non-equivalent path of one FSMD in the other, as there may still be an equivalence because the equivalence checking mechanism in itself is sound but not complete. A path p_1 in an FSMD M_1 is said to contain some path p_2 in another FSMD M_2, if all the statements on the path p_2 are also present in p_1. A formal definition of containment is given in the next section. This containment checking is done in order to look for errors in the FSMD of a student program and do the correction appropriately. This method is discussed in further detail in this chapter.

The remaining of this chapter is organized as follows. In section 2.2, we explain containment checking mechanism with the help of an example and interpretation of its output. In section 2.3 through 2.3.4, we describe the strategies for various types of errors. Finally we conclude the chapter with results.

2.2 Containment checking

In this section we will see the working of containment checking mechanism. The containment checking mechanism is invoked after equivalence checker fails to find equivalence between two FSMDs. Equivalence checker reports before exiting, the *last correct state* (LCS), which is the state beyond which the equivalence checker fails to find equivalence because of various errors in the code such as dependency violation, missing code, skipping parenthesis, discussed in detail in section 2.3. The containment types are broadly of the categories as follows (discussed at length in section 2.2.2): *i)* unordered path-wise both way contained, *ii)* path-wise one way contained, *iii)* path-wise un-contained

Containment checking is required separately, because equivalence checker cannot diagnose why an equivalent path can not be found in the other FSMD, for which it has failed. When the equivalence checker fails, it reports , the complete path from the LCS to the last state (cut-point) it encountered after (*exploring all the possible extensions of the paths in a depth first search manner*), the maximum possible extension of the last path it did in the search. With the help of containment checking mechanism, we are able to analyze the reason by finding the containment of the initial portion of the failing path, for which equivalence is not found in the other FSMD. The initial portion of the failing path is the path between the LCS and the next cut-point state. An error lies on the initial portion of a failing path, i.e., between the LCS and the next cut-point. Containment checking is started at the LCS and it checks the containment of the initial portion of the failing path i.e., the path between the last correct state and the next cut-point state, *inside the path on the correct FSMD which starts from the corresponding state on the correct FSMD of the LCS and the next state on the correct FSMD. If containment is not found then the path selected on the correct FSMD is further extended up to the next cut-point state. Again containment checking is done for the initial portion of the failing path inside the extended path on the correct FSMD. Again on failure, the path on the corrected FSMD is extended to the next state, containment is checked and the process is repeated till the path on the correct FSMD can be extended in a depth first manner, if required backtracking and further extension is done on the other branch till the containment is ascertained to exist or not.* In case there are more than one branches emanating from the LCS, then containment is checked for the initial paths (from the LCS to the next state) on all the branches in a depth first manner as described above, i.e., first checking the containment inside the correct FSMD on the path from the state on the correct FSMD corresponding to the LCS to the next state, extending this path to further next state on the correct FSMD in case the containment was not found and so on. If one of the branches from the LCS on the incorrect FSMD were found equal and the remaining were only contained, then such a case is termed as unordered path-wise both way contained and path-wise one way contained for faulty branching in this chapter. Unordered path-wise both way contained means all the data transformations and conditions are equal on both the FSMDs, their order of occurrence may be different. Path-wise one way contained means that all the data transformation and conditions in the incorrect FSMD are also present in the correct FSMD, (which has some more of them) their order of occurrence may be different. Path-wise un-contained means none of the the data transformations and conditions beyond LCS to the next cut-point are equal on both the FSMDs. Detailed examples of above cases are given further in this section.

The algorithm used for containment checking has been given later in this section. Before that the following definitions are in order.

Definition 8 (Last correct state (LCS) and its corresponding state)**.** *Last correct state (LCS) is defined as the cut-point state in M_s, from which there is a path, for which the equivalence checker could not find an equivalent path (in M_g).*

The cut-point state in the correct FSMD M_g, which corresponds to the LCS, is *called the corresponding state of the last correct state (CSLCS).*

For containment checking we may have to extend paths. We now motivate the definition of an extended path. Suppose we are trying to find the containment of all the statements of a program, say M_s, which are along the path ξ_s, within the path χ_g. In other words, we want to find whether all the program statements along the path ξ_s are also present on the path χ_g or not. Suppose the containment was not found. We then extend χ to χ_e, such that $\chi_e = \chi \bullet \chi'$ where χ' is a next concatenated path such that, $\chi'^s = \chi^f$ and $(\chi')^f$ is a cut-point. Extending this way several times, we may find that χ_e contains ξ_s.

Definition 9 (Containment of a path from a given cut point)**.** *A path p_s of an FSMD M_s from its LCS= p_s^s, is said to be contained in a path or an extended path p_g of the FSMD M_g with p_g^s=CSLCS, if for every transition $t_{s,i} : \langle C_i/l_i \leftarrow e_i(\overline{v})\rangle$ in p_s, there exists a transition $t_{g,j} : \langle C_j/l_j \leftarrow e_j(\overline{v})\rangle$ in p_g, such that $(l_i = l_j) \wedge (e_i(\overline{v}) \cong e_j(\overline{v}))$. Equivalently, p_g is said to contain p_s.*

It may be noted that containment is of a syntactic nature only, but not with respect to the symbolic data transformation, on account of which the equivalence checker reports failure, in the first place.

Illustrative examples for various types of containment

In the following example the case of *unordered path-wise both way contained and path-wise one way contained for faulty branching* occurs. This case is applicable when the closing parenthesis is wrongly put beyond its proper place in the code by the student. For example, if the closing parenthesis of the first if-else block has been put at the end of the second if-else block in a program where two consecutive if-else blocks are present. An error like this in the code causes one of the transitions emanating from the starting state of first if-else block, to skip meeting the final state of this block and instead be incident on the final state of the next if-else block. We refer to this type of error as that of *skipping parenthesis*. This example discusses such a case.

Example 2.1. Let us consider the two FSMDs given in the figures 2.1 and 2.2. Their equivalence checking starts with the respective starting states, q_a and q_{00}, assumed

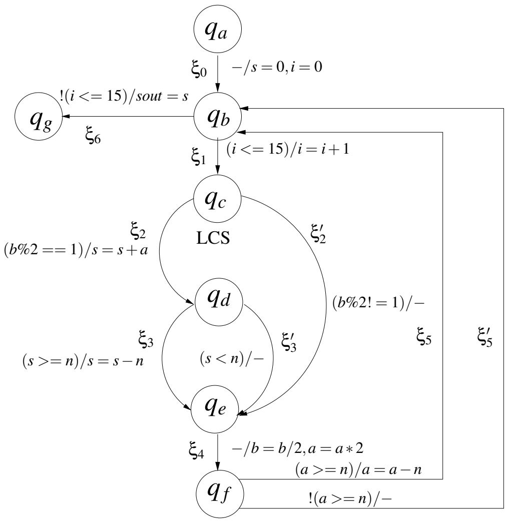


Figure 2.1: FSMD of student's program

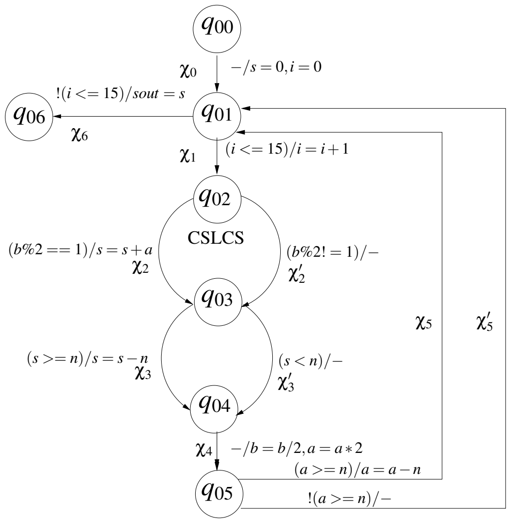


Figure 2.2: FSMD of golden program

to be corresponding states. Starting from this pair of corresponding states, the paths to the next cut-point in both the FSMDs are checked and found to be equivalent, i.e., $q_a \rightarrow q_b \cong q_{00} \rightarrow q_{01}$. Now q_b and q_{01} become corresponding states. The path $q_b \rightarrow q_g$ (corresponding transition ξ_6) is checked next and found to have an equivalent path $q_{01} \rightarrow q_{06}$ (χ_6). The other path from q_b viz., $q_b \rightarrow q_c$ (ξ_1) is found to have an equivalent path $q_{01} \rightarrow q_{02}$ (χ_1) making q_c and q_{02} corresponding states. Let the next path chosen from q_c be $q_c \rightarrow q_d$ (ξ_2) which is found to have the equivalent path $q_{02} \rightarrow q_{03}$ (χ_2) in the FSMD of figure 2.2. The next path from q_c is $q_c \rightarrow q_f$ ($\xi_2'\xi_4$). No equivalent paths from the corresponding cut-point state q_{02} is found for ($\xi_2'\xi_4$). The path $q_c \rightarrow q_f$ ($\xi_2'\xi_4$) is further extended to the next cut-point q_b. As q_f has two transitions emanating from it, the extension is done through both. In both cases, i.e., $\xi_2'\xi_4\xi_5$ and $\xi_2'\xi_4\xi_5'$, the equivalence checker fails to find equivalent paths in the FSMD of figure 2.2. The two paths $q_c \rightarrow q_b$ ($\xi_2'\xi_4\xi_5$ and $\xi_2'\xi_4\xi_5'$) are now extended in turn to the next cut-point q_c as $\xi_2'\xi_4\xi_5\xi_1$ and $\xi_2'\xi_4\xi_5'\xi_1$; in both the cases, no equivalent path is found in the FSMD of figure 2.2. The extended paths have their start state and final state same viz., state q_c, indicating a loop. Hence they are not extended through $q_c \rightarrow q_d$ or $q_c \rightarrow q_e \rightarrow q_f$. The two paths $q_c \rightarrow q_b$ i.e., $\xi_2'\xi_4\xi_5$ and $\xi_2'\xi_4\xi_5'$, are then extended in turn to the next cut-point q_g as $\xi_2'\xi_4\xi_5\xi_6$ and $\xi_2'\xi_4\xi_5'\xi_6$. In this case also, no equivalent path is found in the FSMD of figure 2.2. As no more extension of the path is possible and as all the possible paths emanating from q_c have been checked with extension applied in case of failures, the equivalence checker terminates reporting the extended path $q_c \rightarrow q_g$, for which equivalence could not be detected. As q_c is the starting state of the failed path $q_c \rightarrow q_g$, the state q_c is fed to the containment checker, which then takes over to discover the containment type.

Now we explore the containment checking scenario in the two figures starting from the state q_c and the state having correspondence with the state q_c. In containment checker parlance the state q_c is referred to as LCS and the corresponding state q_{02} is referred to as CSLCS. Again assume that the containment checker starts with checking the containment of the path $q_c \rightarrow q_d$ (ξ_2) in the FSMD of figure 2.2 and finds that this path is statement-wise equivalent to the path $q_{02} \rightarrow q_{03}$ (χ_2) in the FSMD of figure 2.2. Then the containment checker picks up the other path $q_c \rightarrow q_f$ ($\xi_2'\xi_4$), from q_c to the next cut-point q_f. It first compares this path with the two paths emanating from q_{02} to the next cut-point q_{03} (χ_2 and χ_2'), one by one and finds that none of them contains it. Hence, the containment checker extends the paths further to the next cut-point q_{05} and this time it finds that the path $q_{02} \xrightarrow{\chi_2'\chi_3'\chi_4} q_{05}$ actually contains the path $q_c \xrightarrow{\xi_2'\xi_4} q_f$ with the transition equalities $\xi_2' = \chi_2'$ and $\xi_4 = \chi_4$.

As the containment checker found that one of the paths emanating from it has an equal path ($\xi_2 \equiv \chi_2$) in the FSMD of figure 2.2 and the other emanating path is contained ($\chi_2'\chi_3'\chi_4$ contains the path $\xi_2'\xi_4$) in the FSMD of figure 2.2, we call this type of containment as “unordered path-wise both way contained and path-wise one way contained for faulty branching”, which occurs in case of parenthesis skipping. This example may also be used to understand the case “unordered path-wise both way contained” and the case “path-wise one way contained”, which occur in case of error of dependency violation and the error of missing code respectively. We briefly introduce the notion of dependency violation here, although it is discussed at length with an example in the section 2.3.1. A student program is said to have dependency violation error, if two statements in which a statement that has dependency on the other, is written before the statement on which it depends. In case of dependency violation, the containment is “unordered path-wise both way contained”. The error handling, in case of dependency violation is done by reordering the statements in the code according to their order in the golden model. This case has been described later in this thesis. The containment obviously is “path-wise one way contained” in this case of missing code. The case of missing code is handled by introducing missing states and transitions one at a time, again finding the mismatch by equivalence checking and containment checking, and correcting further by introducing other states and transitions and so on till the golden and the student FSMDs become similar. We saw in this example that in the case of just the error of parenthesis skipping, there need be no dependency violation or an instance of missing code, yet the containment is “unordered path-wise both way contained and path-wise one way contained ”. The containment for parenthesis skipping is therefore written as “unordered path-wise both way contained and path-wise one way contained for faulty branching”. We conclude the discussion by observing that the modified FSMD structure (i.e., the FSMD of student’s program) in the two errors of dependency violation and a case of missing code is similar in part to the case presented in this example.

Another example having the outline of containment checking is given below. This example is an example of the case where containment is “path-wise one way contained”.

Example 2.2. We consider two FSMDs as in the figures 2.3 and 2.4,

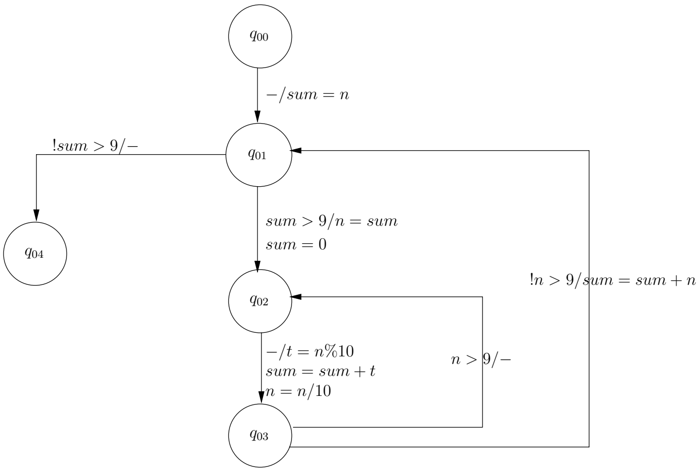


Figure 2.3: M_g, path-wise one way contained example

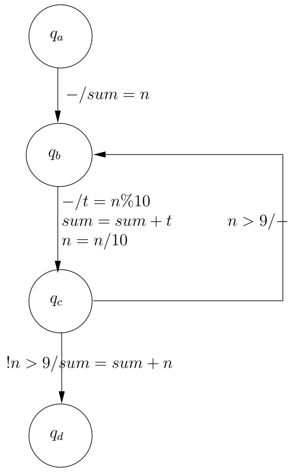


Figure 2.4: M_s, path-wise one way contained example

Clearly, the FSMDs are not equivalent. The equivalence checker reports them to be not equivalent. The LCS and CSLCS are reported to be the states q_a and q_{00} respectively as the equivalence checker finds no equivalence when it starts to examine equivalence from these start states and tries to find the equivalent paths from these start states to the next cut-points in the respective FSMDs.

Now we explore the containment checking scenario in the two figures. The path $p_1 : q_a \rightarrow q_b \rightarrow q_c$ of the incorrect FSMD M_s of figure 2.4 has its starting state p_1^s $= q_a$. p_1 is a path starting from the cut-point q_a to the next cut-point q_c. The path

$p_0 : q_{00} \rightarrow q_{01} \rightarrow q_{02} \rightarrow q_{03}$ on the correct FSMD M_g of figure 2.3, which is actually an extended path on M_g, as it has been extended beyond the cut-point q_{01}. p_0 has its starting state $p_0^s = q_{00}$. There are only two transitions in p_1, $t_{1,0} : q_a \rightarrow q_b$ and $t_{1,1}$: $q_b \rightarrow q_c$. Transition $t_{1,0}$ of p_1 matches the transition $t_{0,0}$: $q_{00} \rightarrow q_{01}$ of p_0; similarly the transition $t_{1,1}$ of p_1 matches the transition $t_{0,2}$: $q_{02} \rightarrow q_{03}$ of p_0. Hence, p_1 of M_s is contained in the extended path p_0 of M_g. As there are only two transitions in p_1 and three transitions in p_0, p_0 has one extra transition. We have referred to this type of containment as "path-wise one way contained". This containment indicates that there is some extra code in the correct FSMD of figure 2.3, which is missing in the incorrect FSMD of figure 2.4 and the error may be because of this. The extra code between the states q_{01} and q_{02} is informed to the student as the missing code in his program. Later in sections 2.3.3 to 2.3.8, we discuss the details of the schemes to insert the missing code of various types in student's program and correcting the student FSMD in steps to make it equivalent to the golden FSMD.

The following example explains the path-wise un-contained case. Here there are two possibilities

i) If the result of containment checking is path-wise un-contained, while checking the containment of incorrect FSMD within the correct FSMD, then it means that incorrect code has got some extra code which is not present in the correct code.

ii) If the result of containment checking is path-wise un-contained, while checking the containment of correct FSMD within the incorrect FSMD, then it means that incorrect code has some code missing.

Example 2.3. This example depicts the case "path-wise un-contained" with the help of figures 2.5 and 2.6. The FSMDs in this case are shown such that the incorrect FSMD has some code extra.

The equivalence checker fails in this case and the start state of the failed path is q_{03} in the incorrect FSMD (M_s) and the corresponding state in the correct FSMD (M_g) is q_c. Containment checker starts comparing the path $q_{03} \xrightarrow{b\%2 == 1/s = s+a} q_{04}$ with the path $q_c \xrightarrow{s > -n/s = s-n} q_d$ in the other FSMD. The path $q_c \rightarrow q_d$ is then extended and containment is not found. All the possible extensions of the path result in failure. Now containment checker starts comparing path $q_{03} \xrightarrow{b\%2! = 1/-} q_{04}$ with the path $q_c \xrightarrow{s < n/-}$ q_d in the other FSMD. The path $q_c \rightarrow q_d$ is then extended and containment is not

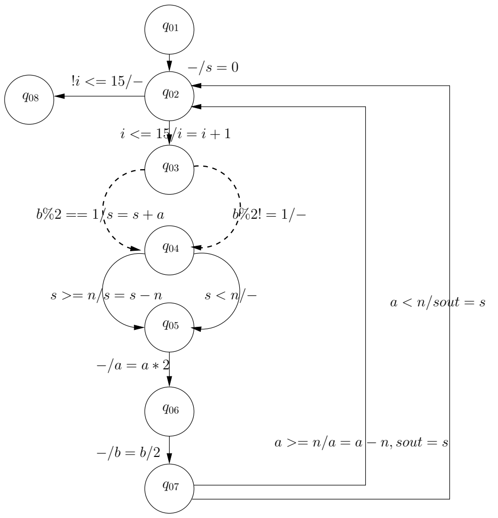


Figure 2.5: M_s, path-wise un-contained example

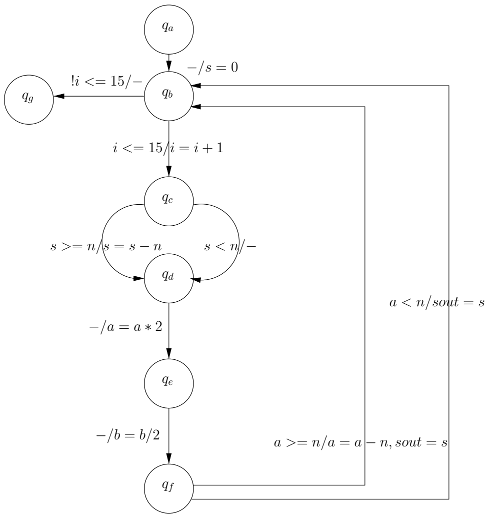


Figure 2.6: M_g, path-wise un-contained example

found. All the possible extensions of the path result in failure. Once again containment checker fails. The containment checker in this case reports "path-wise un-contained" and so extra code is detected between the states q_{03} and q_{04}, which has to be removed. The paths $q_{03} \rightarrow q_{04}$ are now removed, merging the states q_{03} and q_{04}, thus, making the two FSMDs equivalent. The student is informed that the code between the states q_{03} and q_{04} is extra in his program.

Definition 10 (Un-containment of a path from a given cut-point)**.** *A path p_s, starting from LCS, inside the FSMD M_s, is said to be un-contained in the golden FSMD M_g, if all of the transitions of p_s are not found on any path starting from the CSLCS on the golden FSMD M_g.*

2.2.1 Outline of containment checking algorithm

For every path in the incorrect FSMD M_s between the *last correct state* and the next cut-point, but excluding the path which have been successfully handled by the equivalence checker, the algorithm tries to find if that path is syntactically contained in any of the paths of the golden FSMD M_g starting from CSLCS, the state that corresponds to the *last correct state*. In order to find out this containment, at first, any one transition starting from the corresponding state of the *last correct state* is chosen in the correct FSMD and attempt is made to find the containment. If the containment as desired above is not found, then the algorithm tries to find the containment by consecutively extending the path which starts with the transition that was chosen in the previous step until the containment is discovered in a depth first search manner. This may require backtracking (as is done in a depth first search) from terminal state to a previous state and selecting the other transition, if any, from the backtracked state. This method ensures that all the paths to all the reachable states are covered. The containment is established by the fact that the condition and data-transformation in the contained path is a subset of the condition and data-transformation in the containing path. Finding containment for the path $q_c \rightarrow q_d$ of M_s is shown in Figures 2.7 and 2.8. Finding containment for the path $q_c \rightarrow q_f$ of M_s is indicated in Figure 2.9. Extending the path $q_{02} \rightarrow q_{03}$ of M_g for finding containment is shown stepwise in Figures 2.10, 2.11 and 2.12.

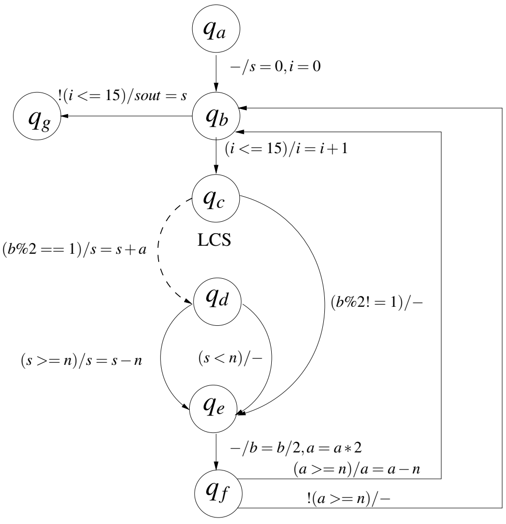


Figure 2.7: M_s: path $q_c \rightarrow q_d$ shown with a dashed line, for which containment is to be found in M_g.

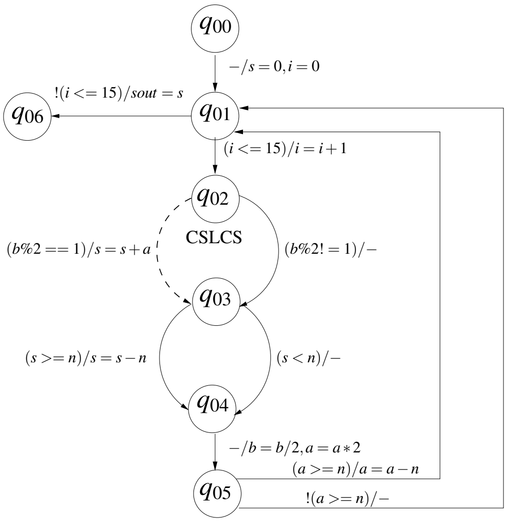


Figure 2.8: M_g: both ways containment of path $q_c \rightarrow q_d$ of M_s found in path $q_{02} \rightarrow q_{03}$, shown with a dashed line.

After containment checking, the type of coding error is identified and then that is handled by a method specific to that error. In the following, handling of dependency violation error is described. In case of dependency violation, the equivalence checker does not find the FSMD of the program having dependency violation, equivalent to the golden program. Subsequently, the containment checker reports *unordered path-wise both way contained*, as there is only a reordering of statements in this case, however, no statement is found missing in the paths along which dependency is violated. The

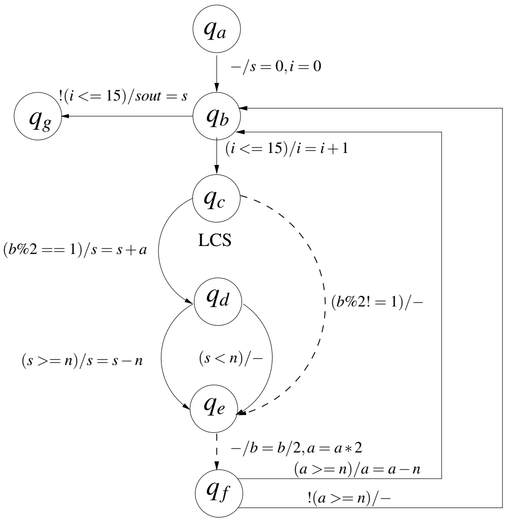


Figure 2.9: M_s: path $q_c \to q_f$ is shown with a dashed line, for which containment is to be found in M_g.

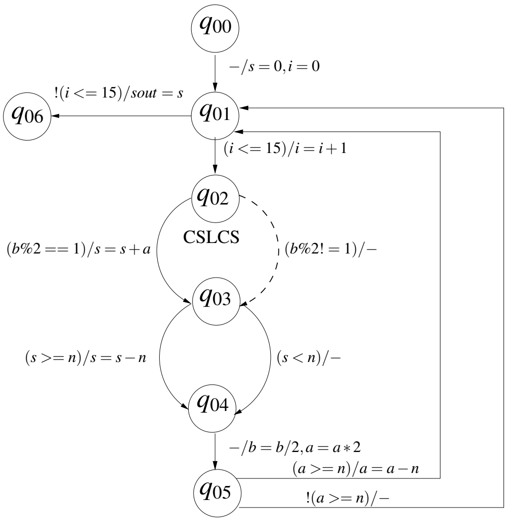


Figure 2.10: M_g: start with path $q_{02} \to q_{03}$, shown with a dashed line.

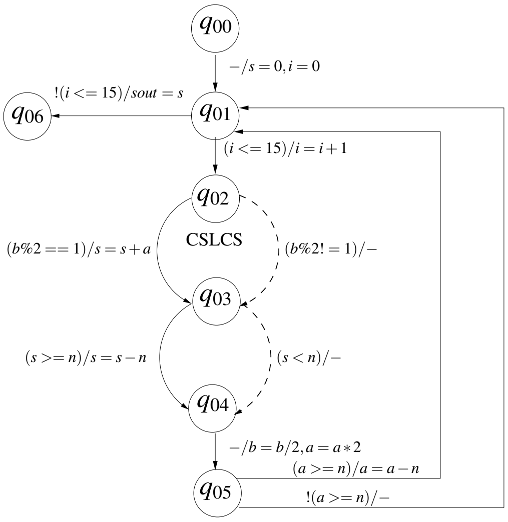


Figure 2.11: M_g: extend the path to q_{04}, shown with dashed lines.

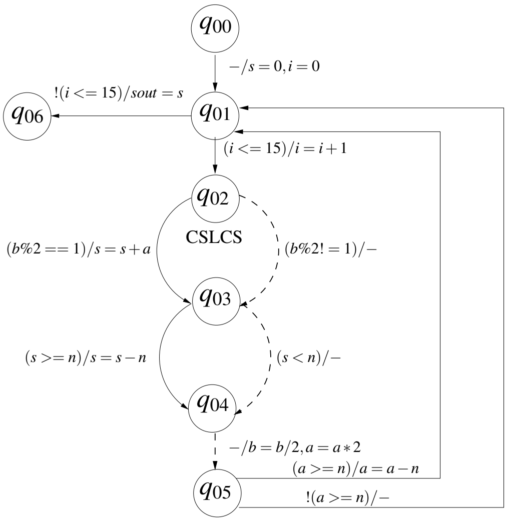


Figure 2.12: M_g: finally containing path $q_{02}...q_{05}$ obtained, shown with dashed lines, by extending to q_{05}.

containment is thus, a both way containment between the paths of the student program having dependency violation and the path of golden program.

Main steps of containment checking:

1. Run the equivalence checker and find the LCS.
2. Choose a path (which has not been checked earlier) starting from the LCS and going up to the next cut-point and execute the following searching steps. Initialize the stacks, *i) pathCovered* (this stack is used to store the subsequently visited transition while proceeding along a path to find the containment and pop the transition while backtracking) and *ii) visited* (this stack is used to store the subsequently visited states while proceeding along a path to find the containment and pop the states while backtracking).
3. Find CSLCS, the state in the correct FSMD, which corresponds to the LCS and push it into the stack *visited.*
   (a) If a path is remaining in correct FSMD starting from the state at the top of the stack *visited*, pick it up and push it to *pathcovered* and go to *step 3b*. If not, pop the stack *visited*, go to *step 3d* (the backtracking step).
   (b) Push the state at the other end of this path into *visited* and check whether the path chosen currently from the incorrect FSMD is contained in the path *pathCovered.* If so, go to *step 4* (exit step). If it is none of the above then go to *step 3c.*
   (c) Pop the stack *visited.* If the popped out state is already present in the stack *visited* or it is the end state of FSMD then go to *step 3d*, else push the popped out state into the stack *visited* and go to *step 3a*.
   (d) Pop the stack *pathCovered.* If the stack *visited* is empty goto step 5, else go to *step 3a*.
4. Here we check the containment in the reverse direction to establish the equality of paths by checking whether the path *pathCovered* is contained in the path chosen currently from the incorrect FSMD. If so, the paths are equal; hence, report the path chosen in incorrect FSMD as unordered path-wise both way contained, else report path-wise one way contained. If some path is remaining between LCS and next cut point in incorrect FSMD then go to *step 2*, else exit.

5. Report the path chosen in incorrect FSMD as path-wise un-contained. If some path is remaining between LCS and next cut point in incorrect FSMD then go to *step 2*, else exit.

Working of containment checking algorithm with an example

We explain the working of the containment checking algorithm with the following example.

Example 2.4. We consider parts of two FSMDs M_g and M_s as given in the Figure 2.13 from LCS and CSLCS onwards. In the Figure, the conditions and data-transformations along the transitions are shown as *cond* and *dtrans* with appropriate subscripts. it may be noted that some of the conditions and data-transformations are identical in M_s and M_g. Those conditions and data-transformations, which are not the same have been shown with different subscripts. The cut-point to cut-point paths are shown as p_s and p_g with appropriate subscripts.

The containment checking proceeds as follows. The containment checker tries to find containment of the path $p_{s,1}$, i.e., $q_c \rightarrow q_d$ of M_s, which is from LCS to the next cut-point, inside the path $p_{g,1}$ i.e., $q_{02} \rightarrow q_{03}$ of M_g, which is from CSLCS to the next cut-point. As the condition and data-transformations along the two paths are same, hence containment is found. The containment checker reports $p_{s,1}$ "Unordered path-wise both way contained" in path $p_{g,1}$. Both these paths are now excluded for further checking by the containment checker. Now the containment checker backtracks from q_{03} to q_{02} and tries to find the containment of the remaining path $p_{s,2}$ from LCS, in the remaining path $p_{g,2}$ emanating from CSLCS, i.e., q_{02}. This time containment is not found due to different data-transformation expressions. The path $p_{g,2}$ is, therefore, extended to the next cut-point q_{05} along one of the paths emanating from q_{03} , say $p_{g,3}$. As none of the data-transformations along the path $p_{g,3}$ is same as that of $p_{s,1}$, hence, the containment is not found. The containment checker now backtracks to the state q_{03} to explore the other path emanating from it, by extending along $p_{g,4}$. This time again the containment is not found for the similar reason as before. The containment checker, therefore backtracks to q_{03}, and as there is no other path from it, hence it further backtracks to q_{02}. As all the paths from q_{02} have been explored and as nowhere the containment was found, hence the containment checker reports $p_{s,2}$ as "path-wise un-contained".

Algorithm 1: Containment_checker

```
    Input: $M_1$, FSMD of program 1 and $M_2$, FSMD of program 2;
    Output: Containment checking status;
L1  [Step 1:] Execute the equivalence checker with the following parameters: (i)$M_1$ and (ii)$M_2$
L2  if equivalence checker reports $M_1 \not\sqsubseteq M_2$ then let $q_1$ be the LCS of FSMD $M_1$ and $q_2$ be the CSLCS in the FSMD $M_2$;
L3  for each path $p'_2$ of $M_2$, starting from one cut-point to the next do
L4      $marked[p'_2] \leftarrow False$; $traversed[p'_2] \leftarrow False$;
    /*Outer for loop for paths from $M_1$                                        */
L5  for each path $p_1$ of $M_1$, starting from $q_1$ do
L6      [Step 2:] Initialize the stacks, (i) pathCovered and (ii) visited
L7      for each cut-point $q_{c_2}$ of $M_2$, starting from CSLCS do
L8          $inVisited[q_{c_2}] \leftarrow False$;
L9      [Step 3:] $visited \leftarrow$ push($visited$, $q_2$); $inVisited[q_2] \leftarrow True$; /*push $q_2$ into the stack visited*/
        /*Inner while loop for paths from $M_2$                                  */
L10     while notEmpty(visited) do
L11         [Step 3a:]
L12         if $p_2$, a path of $M_2$, starting from the state at the top of the stack visited, $q_{2,visitedTop}$, is available s.t.
              traversed[$p_2$]==False && marked[$p_2$]==False then
L13             $traversed[p_2] \leftarrow True$;
L14             if IsEmpty(pathCovered) then
L15                 $p_{2_{extended}} \leftarrow p_2$
L16             else
L17                 $p_{2_{extended}} \leftarrow top(pathCovered) \bullet p_2$
L18             $pathCovered \leftarrow$ push($pathCovered$, $p_{2_{extended}}$)
L19         else
L20             $q_{2_{tm}} \leftarrow pop(visited)$; $inVisited[q_{2_{tm}}] \leftarrow False$;
L21             for each path $p_{2_{tm}}$ of $M_2$, starting from $q_{2_{tm}}$ to the next cut-point do
L22                 $traversed[p_{2_{tm}}] \leftarrow False$
L23             go to L33, step 3d (backtracking step)
L24         [Step 3b:] $visited \leftarrow$ push($visited$, $p^f_{2_{extended}}$); $inVisited[p^f_{2_{extended}}] \leftarrow True$, $p^f_{2_{extended}}$ is the state at the other
            end of $p_{2_{extended}}$;
L25         if the path $p_1$ is contained in $p_{2_{extended}}$ then
L26             break; /* go to L37, step 4 (exit step) */
L27         [Step 3c:] $q_{2_{tmp}} \leftarrow pop(visited)$ ;
L28         if $inVisited[q_{2_{tmp}}] == True \vee q_{2_{tmp}}$ is an end state of $M_2$ then
L29             $inVisited[q_{2_{tmp}}] \leftarrow False$;
L30         else
L31             $visited \leftarrow push(visited, q_{2_{tmp}})$; continue
L32         [Step 3d:]
L33         if $IsEmpty(pathCovered) == False$ then
L34             $p_{2_{tmp}} \leftarrow pop(pathCovered)$
L35         if $IsEmpty(visited) == True$ then
L36             go to L43, step 5 (exit step)
L37     [Step 4:] $marked[p_{2_i}] \leftarrow True$, where $p_{2_i}$ is the sub-path of $p_{2_{extended}}$, s.t. $p^s_{2_i} == q_2$ and $p^f_{2_i}$ is the cut-point next
        to $q_2$ along $p_{2_{extended}}$;
L38     if $p_{2_{extended}}$ is contained in $p_1$ then
L39         the paths are equal; report the path $p_1$, chosen in $M_1$ as unordered path-wise both way contained, report
              $p_{2_{extended}}$
L40     else
L41         report the path $p_1$, chosen in $M_1$, as path-wise one way contained, report $p_{2_{extended}}$
L42     continue;
L43     [Step 5:] report the path $p_1$, chosen in $M_1$, as path-wise un-contained.
```

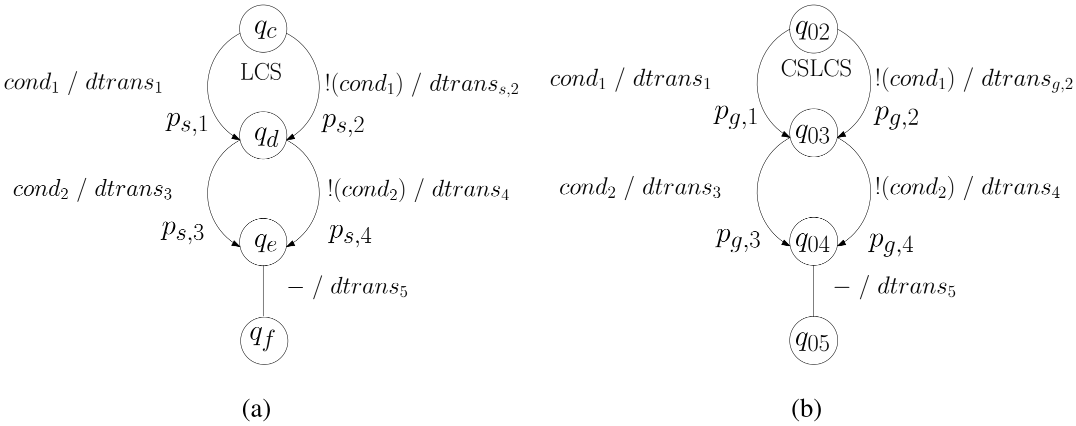


Figure 2.13: Parts of FSMDs (a) M_s and (b) M_g shown from LCS and CSLCS onwards.

The step-by-step containment checking procedure according to the containment checking algorithm is as follows. The evolution of stacks in the example run of the algorithm for containment checking is also shown in each step, where necessary.

Step 1:

LCS: $q_s \leftarrow q_c$;

CSLCS: $q_g \leftarrow q_{02}$;

$\forall p'_g \in M_g, marked[p_g] \leftarrow False$;

$\forall p'_g \in M_g, traversed[p_g] \leftarrow False$;

Beginning of for loop:

$p_s \leftarrow p_{s,1}$

Step 2:

Initialize the stacks *pathCovered* and *visited*;

$\forall$ cut-points q_{c_g} of M_g, starting from CSLCS, $inVisited[q_{c_g}] \leftarrow False.$

$\boxed{-}$ pathCovered $\boxed{-}$ visited

Step 3:

$visited \leftarrow push(visited, q_{02})$

$inVisited[q_{02}] \leftarrow True$

$\boxed{-}$ pathCovered $\boxed{q_{02}}$ visited

Beginning of while loop:

$notEmpty(visited) == True$

Step 3a:

$p_g \leftarrow p_{g,1}$

$traversed[p_{g,1}] \leftarrow True$

$IsEmpty(pathCovered) == True$

$p_{g_{extended}} \leftarrow p_{g,1}$

$pathCovered \leftarrow push(pathCovered, p_{g,1})$

$\boxed{p_{g,1}}$ pathCovered $\boxed{q_{02}}$ visited

Step 3b:

$p^f_{g_{extended}} \leftarrow q_{03};$

$visited \leftarrow push(visited, q_{03});$

$inVisited[q_{03}] \leftarrow True;$

p_s is contained in $p_{g,1}$, so

p_s is contained in $p_{g_{extended}}$, goto step 4

$$\begin{array}{|c|} p_{g,1} \\ \hline \end{array}\text{ pathCovered} \qquad \begin{array}{|c|} q_{03} \\ q_{02} \\ \hline \end{array}\text{ visited}$$

Step 4:

$marked[p_{g,1}] \leftarrow True$

$p_{g,1}$ is contained in $p_{s,1}$, so

$p_{g_{extended}}$ is contained in p_s

Report: $p_{s,1}$ is unordered path-wise both way contained

continue;

$$\begin{array}{|c|} p_{g,1} \\ \hline \end{array}\text{ pathCovered} \qquad \begin{array}{|c|} q_{03} \\ q_{02} \\ \hline \end{array}\text{ visited}$$

Beginning of for loop:

$p_s \leftarrow p_{s,2}$

Step 2:

Initialize the stacks pathCovered and visited.

$\forall$ cut-points q_{c_g} of M_g, starting from CSLCS, $inVisited[q_{c_g}] \leftarrow False$.

$$\begin{array}{|c|} - \\ \hline \end{array}\text{ pathCovered} \qquad \begin{array}{|c|} - \\ \hline \end{array}\text{ visited}$$

Step 3:

$visited \leftarrow push(visited, q_{02})$

$inVisited[q_{02}] \leftarrow True$

$$\begin{array}{|c|} - \\ \hline \end{array}\text{ pathCovered} \qquad \begin{array}{|c|} q_{02} \\ \hline \end{array}\text{ visited}$$

Beginning of while loop:

$notEmpty(visited) == True$

Step 3a:

Now $p_g \leftarrow p_{g,2}$; // as $p_{g,1}$ is already marked previously in step 4.

$traversed[p_{g,2}] \leftarrow True;$

$IsEmpty(pathCovered) == True;$

$p_{g_{extended}} \leftarrow p_{g,2};$

$pathCovered \leftarrow push(pathCovered, p_{g,2})$

$$\begin{array}{|c|}p_{g,2}\\ \hline\end{array}\ \text{pathCovered} \qquad \begin{array}{|c|}q_{02}\\ \hline\end{array}\ \text{visited}$$

Step 3b:

$p^f_{g_{extended}} \leftarrow q_{03};$

$visited \leftarrow push(visited, q_{03});$

$inVisited[q_{03}] \leftarrow True;$

$p_{s,2}$ is not contained in $p_{g,2}$,

$$\begin{array}{|c|}p_{g,2}\\ \hline\end{array}\ \text{pathCovered} \qquad \begin{array}{|c|}q_{03}\\ q_{02}\\ \hline\end{array}\ \text{visited}$$

Step 3c:

$q_{g_tmp} \leftarrow q_{03} \leftarrow pop(visited);$

The else part is executed in this case.

$visited \leftarrow push(visited, q_03)$; continue

The stacks at the end of the operations in this step are shown as follows.

$$\begin{array}{|c|} p_{g,2} \\ \hline \end{array}\ \text{pathCovered} \qquad \begin{array}{|c|} q_{03} \\ q_{02} \\ \hline \end{array}\ \text{visited}$$

Beginning of while loop:

$notEmpty(visited) == True$

Step 3a:

$q_{03} \leftarrow top(visited);$

Let $p_g \leftarrow p_{g,3};$

$traversed[p_{g,3}] \leftarrow True;$

$IsEmpty(pathCovered) == False;$

$p_{g_{extended}} \leftarrow p_{g,2}, p_{g,3};$

$pathCovered \leftarrow push(pathCovered, \{p_{g,2}, p_{g,3}\})$

$$\begin{array}{|c|} p_{g,2}, p_{g,3} \\ p_{g,2} \\ \hline \end{array}\ \text{pathCovered} \qquad \begin{array}{|c|} q_{03} \\ q_{02} \\ \hline \end{array}\ \text{visited}$$

Step 3b:

$p^f_{g_{extended}} \leftarrow q_{05};$

$visited \leftarrow push(visited, q_{05});$

$inVisited[q_{05}] \leftarrow True;$

$p_{s,2}$ is not contained in $p_{g,2}, p_{g,3}$,

$$\begin{array}{|c|} p_{g,2}, p_{g,3} \\ p_{g,2} \\ \hline \end{array}\ \text{pathCovered} \qquad \begin{array}{|c|} q_{05} \\ q_{03} \\ q_{02} \\ \hline \end{array}\ \text{visited}$$

Step 3c:

$q_{g_tmp} \leftarrow q_{05} \leftarrow pop(visited);$

q_{05} is end state, so if part is executed

$inVisited[q_{05}] \leftarrow False;$

q_{05} is end state, so go to step 3d

$$\begin{array}{|c|} p_{g,2}, p_{g,3} \\ p_{g,2} \\ \hline \end{array} \text{ pathCovered } \quad \begin{array}{|c|} q_{03} \\ q_{02} \\ \hline \end{array} \text{ visited}$$

Step 3d:

$IsEmpty(pathCovered) == False;$

$p_{g_{tmp}} \leftarrow p_{g,2}, p_{g,3} \leftarrow pop(pathCovered)$

$IsEmpty(visited) == False$, so while loop continues

$$\begin{array}{|c|} p_{g,2} \\ \hline \end{array} \text{ pathCovered } \quad \begin{array}{|c|} q_{03} \\ q_{02} \\ \hline \end{array} \text{ visited}$$

Beginning of while loop:

$notEmpty(visited) == True$

Step 3a:

$q_{03} \leftarrow top(visited);$

$p_g \leftarrow p_{g,4};$

$traversed[p_{g,4}] \leftarrow True;$

$IsEmpty(pathCovered) == False$; so, else part is executed and path extended

$p_{g_{extended}} \leftarrow p_{g,2}, p_{g,4};$

$pathCovered \leftarrow push(pathCovered, p_{g,2}, p_{g,4})$

$p_{g,2}, p_{g,4}$	pathCovered	q_{03}	visited
$p_{g,2}$		q_{02}	

Step 3b:

$p^{f}_{g_{extended}} \leftarrow q_{05};$

$visited \leftarrow push(visited, q_{05});$

$inVisited[q_{05}] \leftarrow True;$

$p_{s,2}$ is not contained in $p_{g,2}, p_{g,4}$,

		q_{05}	
$p_{g,2}, p_{g,4}$	pathCovered	q_{03}	visited
$p_{g,2}$		q_{02}	

Step 3c:

$q_{g_tmp} \leftarrow q_{05} \leftarrow pop(visited);$

$inVisited[q_{05}] \leftarrow True$ also, q_{05} is end state, so if part is executed

$inVisited[q_{05}] \leftarrow False;$

next is step 3d

$p_{g,2}, p_{g,4}$	pathCovered	q_{03}	visited
$p_{g,2}$		q_{02}	

Step 3d:

$IsEmpty(pathCovered) == False;$

$p_{g_{tmp}} \leftarrow p_{g,2}, p_{g,4} \leftarrow pop(pathCovered)$

$IsEmpty(visited) == False$, so loop again in while loop

$$\begin{array}{|c|}p_{g,2}\\\hline\end{array}\ \text{pathCovered}\qquad\begin{array}{|c|}q_{03}\\q_{02}\\\hline\end{array}\ \text{visited}$$

Beginning of while loop:

$notEmpty(visited) == True$

Step 3a:

$q_{03} \leftarrow top(visited);$

$traversed[p_{g,3}] == True, marked[p_{g,3}] == False,$

also $traversed[p_{g,4}] == True, marked[p_{g,4}] == False,$

hence, for none of the paths from q_{03}, the if part is to be executed.

Hence, else part is executed.

$q_{03} \leftarrow pop(visited);$

$inVisited[q_{03}] \leftarrow False;$

$traversed[p_{g,3}] == False; traversed[p_{g,4}] == False;$

go to step 3d;

$$\begin{array}{|c|}p_{g,2}\\\hline\end{array}\ \text{pathCovered}\qquad\begin{array}{|c|}q_{02}\\\hline\end{array}\ \text{visited}$$

Step 3d:

$IsEmpty(pathCovered) == False;$

$p_{g_{tmp}} \leftarrow p_{g,2} \leftarrow pop(pathCovered)$

$IsEmpty(visited) == False$

$$\begin{array}{|c|}-\\\hline\end{array}\ \text{pathCovered}\qquad\begin{array}{|c|}q_{02}\\\hline\end{array}\ \text{visited}$$

Beginning of while loop:

$notEmpty(visited) == True$

Step 3a:

$q_{02} \leftarrow top(visited);$

$traversed[p_{g,1}] == True, marked[p_{g,1}] == True,$

also $traversed[p_{g,2}] == True, marked[p_{g,2}] == False,$

hence, for none of the paths from q_{02}, the if part is to be executed.

Hence, else part is executed.

$q_{02} \leftarrow pop(visited);$

$inVisited[q_{02}] \leftarrow False;$

$traversed[p_{g,1}] == False; traversed[p_{g,2}] == False;$

go to step 3d;

$\boxed{-}$ pathCovered $\boxed{-}$ visited

Step 3d:

$IsEmpty(pathCovered) == True;$

$IsEmpty(visited) == True$

go to step 5;

$\boxed{-}$ pathCovered $\boxed{-}$ visited

Step 5:

Report: $p_{s,2}$ is path-wise both way un-contained.

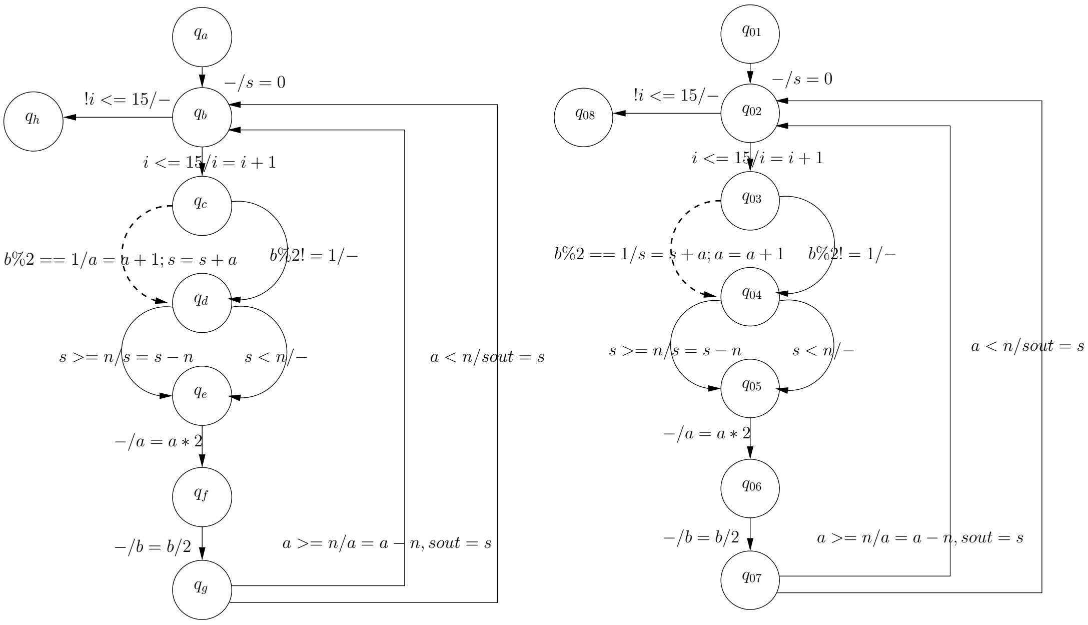
End

⌊–⌋ pathCovered ⌊–⌋ visited

2.2.2 Interpretation of the results of containment checking

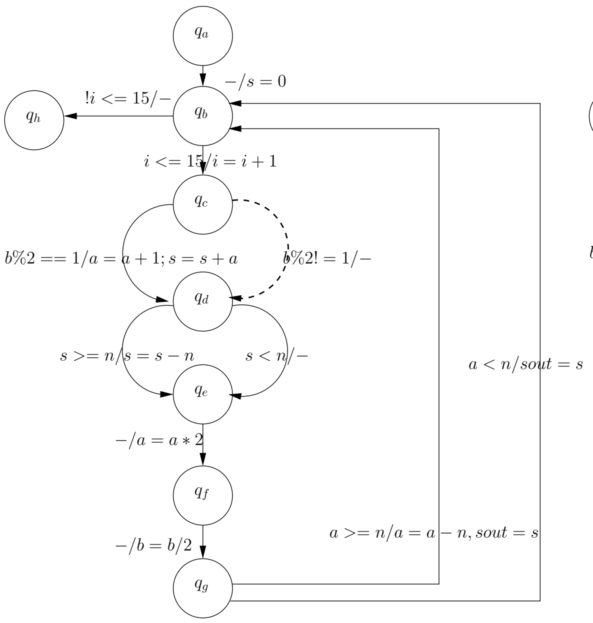
We interpret the results of containment checking through the following cases.

1. *Unordered path-wise both way contained:* If all paths from the LCS to the next cut-point are found unordered path-wise both way contained in their corresponding paths in the correct FSMD as in figures 2.14 and 2.15 and still equivalence checking failed, then this would mean there is dependency violation in the incorrect program.

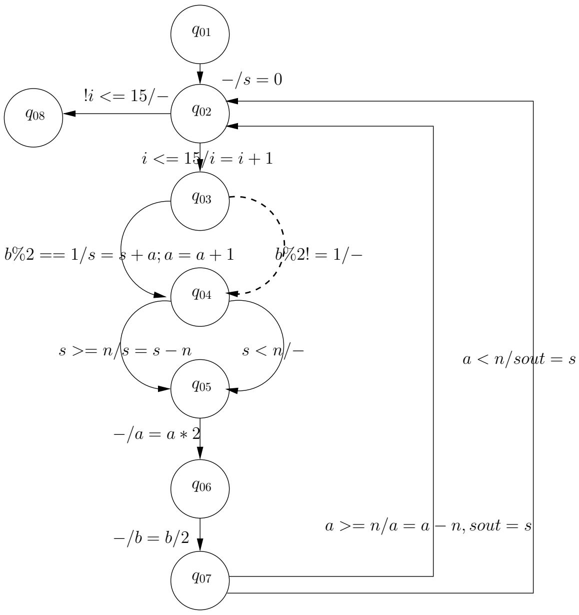


(a) (b)

Figure 2.14: Unordered path-wise both way contained Part-I: FSMD on fig (a) contains the dotted path which is equal to the dotted path shown in the fig (b).



(a)



(b)

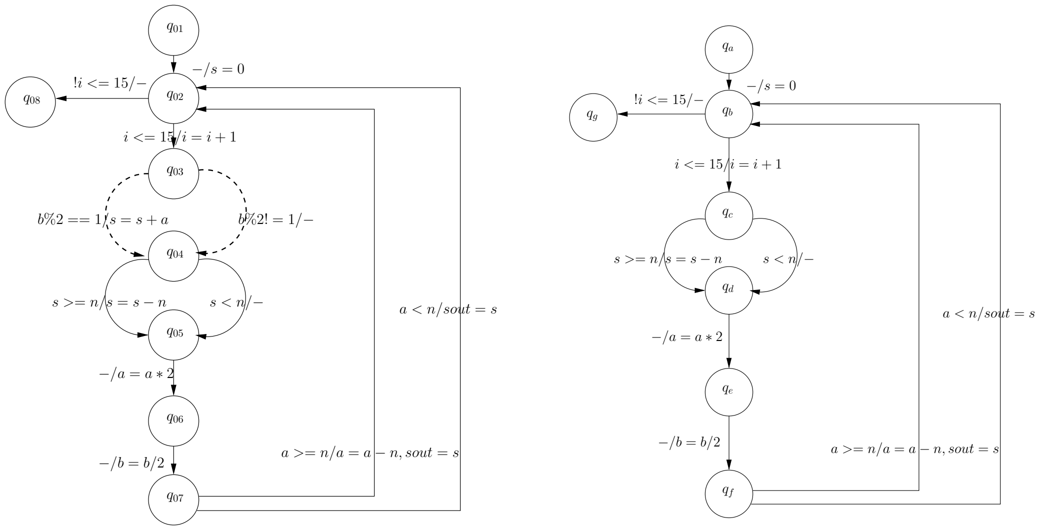
Figure 2.15: Unordered path-wise both way contained Part-II: FSMD on fig (a) contains the dotted path which is equal to the dotted path shown in the fig (b).

2. *Path-wise un-contained:* This case is depicted in the figure 2.16. Here there are two possibilities

   (a) If the result of containment checking is *path-wise un-contained*, while checking the containment of incorrect FSMD within the correct FSMD, then it means that incorrect code has got some extra code which is not present in the correct code.

   (b) If the result of containment checking is *path-wise un-contained*, while checking the containment of correct FSMD within the incorrect FSMD, then it means that incorrect code is missing some code.

3. *Path-wise one way contained:* If the result of containment checking is *path-wise one way contained*, while checking the containment of incorrect FSMD within the correct FSMD, then it means that some extra code before this LCS is missing in the incorrect program. The missing code, therefore, has to be inserted in the incorrect program. The use of correction vector is made in such cases.

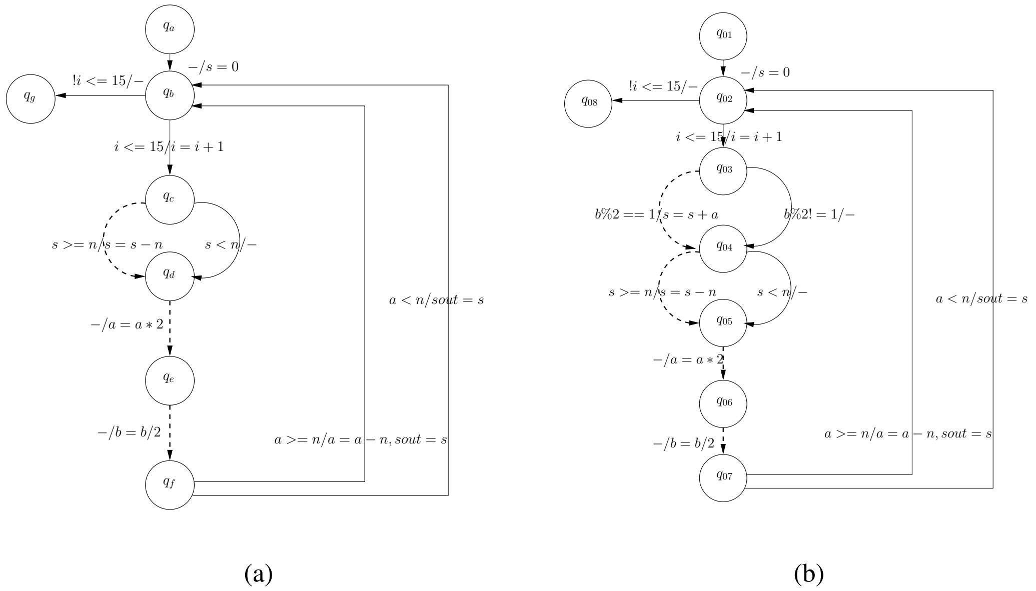


(a) (b)

Figure 2.16: Path-wise un-contained: FSMD in fig (a) contains the dotted paths which are not found in the fig (b) by containment checker.

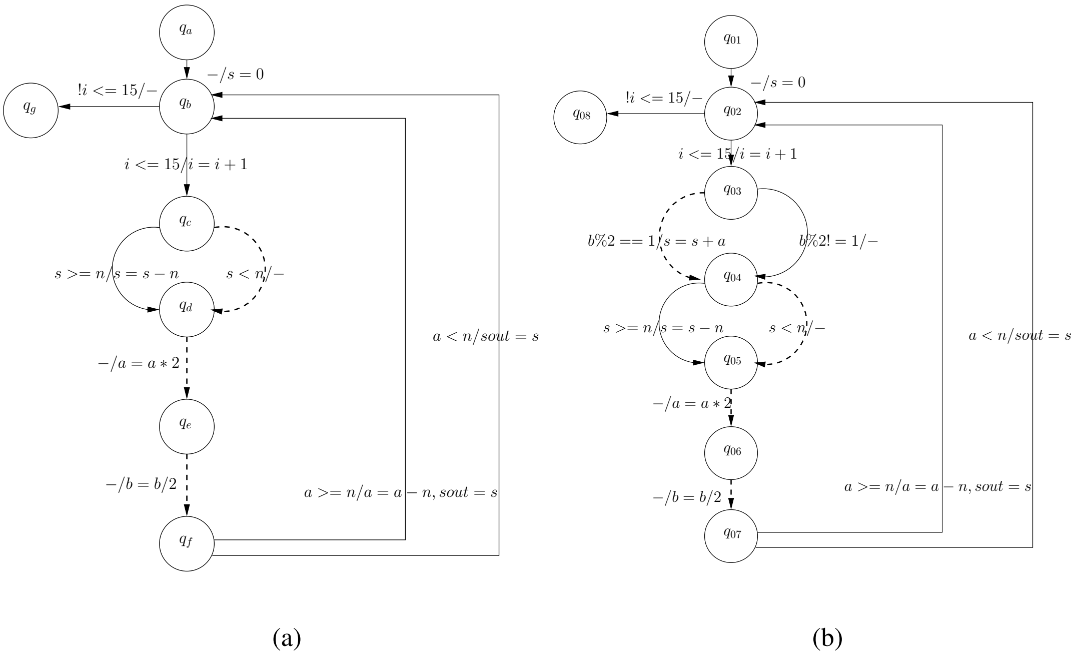
Following figures 2.17 and 2.18 show two examples in which figures on left hand side show paths, which are path-wise one way contained in the FSMD on the right.

4. *Unordered path-wise both way contained and path-wise one way contained for faulty branching:* This case arises when parenthesis is skipped and the resulting edge goes to a state which is not a cut-point. Following figures 2.19 and 2.20 show an example of this case. In the figure 2.19 the path shown in figure (a) is equal to the path shown in figure (b), whereas in the figure 2.20 the path shown in figure (a) is contained in the path shown in figure (b).



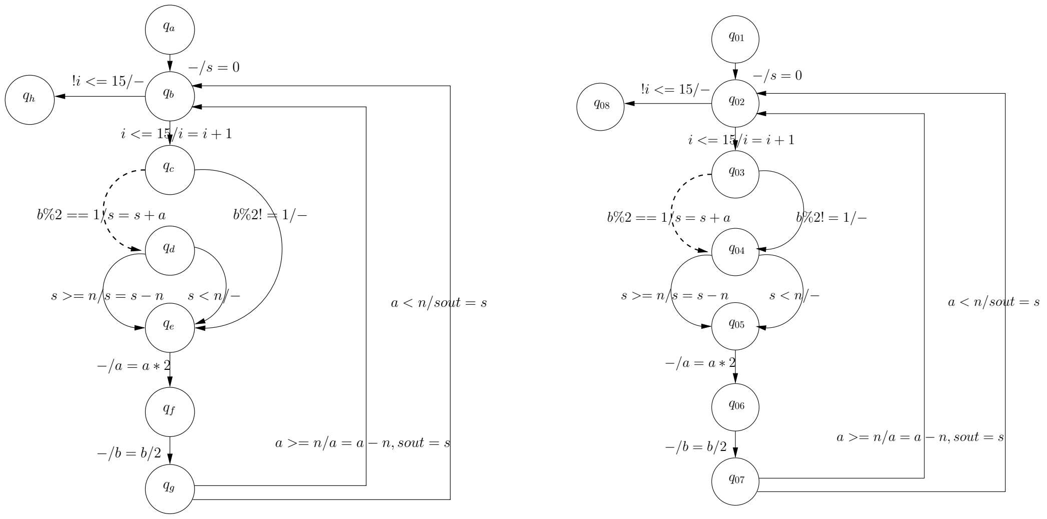
(a) (b)

Figure 2.17: Path-wise one way contained - I: FSMD in fig (a) show paths which are path-wise one way contained in the FSMD in the fig (b).



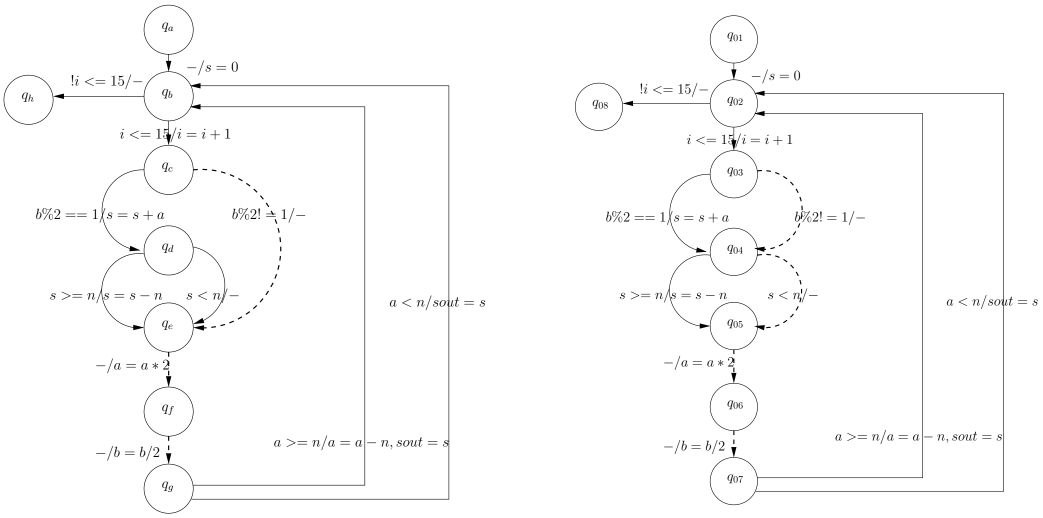
(a) (b)

Figure 2.18: Path-wise one way contained - II: FSMD in fig (a) show paths which are path-wise one way contained in the FSMD in the fig (b).



(a) (b)

Figure 2.19: Unordered path-wise both way contained and path-wise one way contained for faulty branching - I: FSMD in fig (a) shows the path, which is equal to the path shown in the FSMD in fig (b).



(a) (b)

Figure 2.20: Unordered path-wise both way contained and path-wise one way contained for faulty branching - II: FSMD in fig (a) shows the path, which is contained in the path shown in the FSMD in fig (b).

Definition 11 (Containment equivalence of paths)**.** *Two paths p_s in an FSMD M_s and p_g in an FSMD M_g, from a cut-point to the next cut-point are said to be containment equivalent, if containment checker finds them to contain same statements, the statements may occur in different order in the two paths.*

In this section we saw the working of containment checking mechanism. The containment checking mechanism is invoked after the equivalence checker fails to find equivalence. As the equivalence checker cannot find the reason why an equivalent path cannot be found in the other FSMD, because of which then it fails; so, a containment checking mechanism was required. In this section, we analyzed, with the help of containment checking mechanism, the reason of failure of equivalence checking, by finding the containment of the failing path for which equivalence was not found in the other FSMD. In the next section, we give examples of implementations of such analysis with which we are able to differentiate between various types of errors and suggest corrections for them.

We now state some of the limitations of the containment checking algorithm. Firstly, the algorithm for containment checking is an exponential algorithm in the worst-case. The worst-case exponential nature is because of the cut-points. As a cut-point has a minimum of two outward transitions, there are thus at least two possibilities for search path for finding containment. The containment checking algorithm is therefore exponential in the number of such cut-points, as for k cut-points there will exist 2^k paths for finding containment (see figure 2.21) in the worst-case.

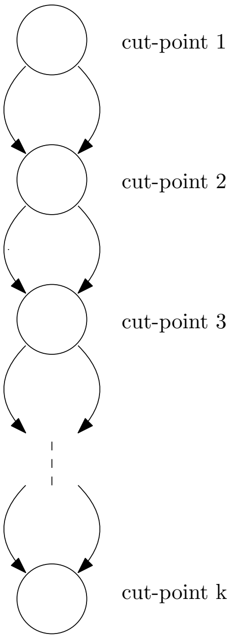


Figure 2.21: Worst-case exponential nature of the containment checking algorithm.

Secondly, as the equivalence checker does not work, when some loop is given in unrolled form or any data-transformation that might cross loop boundary and as the containment checker uses equivalence checker, so this approach will not work in the cases of an unrolled loop or any data-transformation that might cross loop boundary.

2.3 Implementations of strategies for various types of errors

In this section we have tried to classify some of the errors and implemented common strategies for the problems of various such classes. All the strategies given in this chapter involve containment checking. The other strategies discussed are e.g., to adjust the incoming edges at a given state, applying correction vector to introduce missing code, introduction of new states and checking dependency list . At first we identify the types of errors which are discussed below.

Identified types of error

- Dependency has been violated between instructions.
- Skipping Parenthesis
  - Edge corresponding to skipped parenthesis goes to a cut-point state.
  - Edge corresponding to skipped parenthesis goes to a non cut-point state.
  - Edge belongs to loop.
- Some code is missing.

2.3.1 Error of dependency violation

The dependency is said to be violated in a student program, as compared to the golden program, if it has the reverse occurrence of two statements, which should have a precede-succeed ordering. As an example we see the following order of statements in two programs as shown below.

Listing 2.1: code 1

```
void code()
{
    .....
    a=a+1;
    s=s+a;
    .....
}
```

Listing 2.2: code 2

```
void code()
{
    .....
    s=s+a;
    a=a+1;
    .....
}
```

Code in the two programs has reverse order of occurrence of statements, in which s is dependent on value of a. In the first code a is updated first and then used for updating s, whereas in the second code s is updated prior to updating a. Thus the dependency is violated by the second code. All the dependency types, viz., read after write, write after read and write after write can be taken care of by our scheme.

Below we define the dependency violation formally, in our context. This definition applies to the FSMDs having equal number of transitions, which are same after normalization, but occur in a different order in the other FSMD, so that some dependence is violated.

Definition 12 (Dependency violation). *Let p_g be a path of M_g of the form $\langle t_1, \ldots, t_n \rangle$; let p_s be a path of M_s which originates from the state having correspondence with the start state p_g^s and has the same set of transitions as p_g but in different order. For some $i, 2 \leq i \leq n$, let t_i have a dependence on t_{i-k} , for some $k, 0 < k < i$, in p_g. The path p_s is said to have a dependence violation with respect to p_g if t_i corresponds to t_j , for some $j, 1 \leq j \leq n-1$ and t_{i-k} corresponds to $t_{j+k'}$ for some $k', 0 < k' \leq n-j$.*

Theorem 2 (Detection of dependency violation). *i) Let there be a path or an extended path p_s of the student program M_s for which the equivalence checker could not find an equivalent path in the golden model M_g emanating from the state of M_g corresponding to p_s^s*

ii) Let p_s emanating from p_s^s having correspondence with p_g^s be identified to be satisfying both way containment with p_g.

Then, there exists a dependency violation in the path p_s of M_s

Proof [by refutation]: *Let the paths p_g of M_g and p_s of M_s be as indicated in the theorem statement and $p_g \not\simeq p_s$.*

Let there be no dependency violation in p_s with respect to p_g. Hence, we have the following two cases:

Case 1 [No dependence between any two transitions of p_g]:

so the set of variables that are modified in p_g are disjoint with the set of read variables in p_g; also no two transitions of p_g write onto the same variable.

Hence, all the permutations of p_g will have same condition of execution and data-transformation and hence will be equivalent with each other. But p_s is a permutation of p_g but not equivalent to p_g. Thus, we have a contradiction.

Case 2 [Some transitions of p_g have dependences]:

Let $\underline{i}$ be the minimum value of i for which $t^g_{\underline{i}}$ does not match with $t^s_{\underline{i}}$. So, over the range $[1, \underline{i}-1]$, all the transitions match and hence the dependences, if any, of the subsequence of p_g are retained in the corresponding subsequence p_s.

So, $p_g[1,\ldots,\underline{i}-1] \cong p_s[1,\ldots,\underline{i}-1]$.

Let $t^g_{\underline{i}}$ match with t^s_j for some $j > \underline{i}$. Let p_s be modified to p^m_s by placing t^s_j at the $\underline{i}^{th}$ position of p^m_s and $p^m_s[\underline{i}+1,\ldots,j]$ be replaced by $p_s[\underline{i},\ldots,j-1]$; that is, the transitions in the subsequence $\langle t^s_{\underline{i}},\ldots,t_{j-1}\rangle$ of p_s are shifted right by one position. Let any shifting step be referred to as step S. We claim that

i) $p_s \cong p^m_s$ and

ii) $p_g[1,\ldots,\underline{i}] \cong p^m_s[1,\ldots,\underline{i}]$

Now we have the following two sub-cases:

Sub-case 2.1 [$t^g_{\underline{i}}$ has a dependence on $t^g_{\underline{i}-k}$]:

Since, $t^g_{\underline{i}-k} = t^s_{\underline{i}-k}$, the dependence $t^g_{\underline{i}-k} \rightarrow t^g_i$ corresponds to the dependence $t^s_{\underline{i}-k} \rightarrow t^s_j$. Hence when t^s_j is placed at $t^s_{\underline{i}-k}$ in p^m_s by the above step, the dependence is still not violated. Thus, with the above change in placement of the transition, $p_g[1,\ldots,\underline{i}] = p^m_s[1,\ldots,\underline{i}]$. Also since p^m_s retains all the dependences of p_s, $p_s \cong p^m_s$.

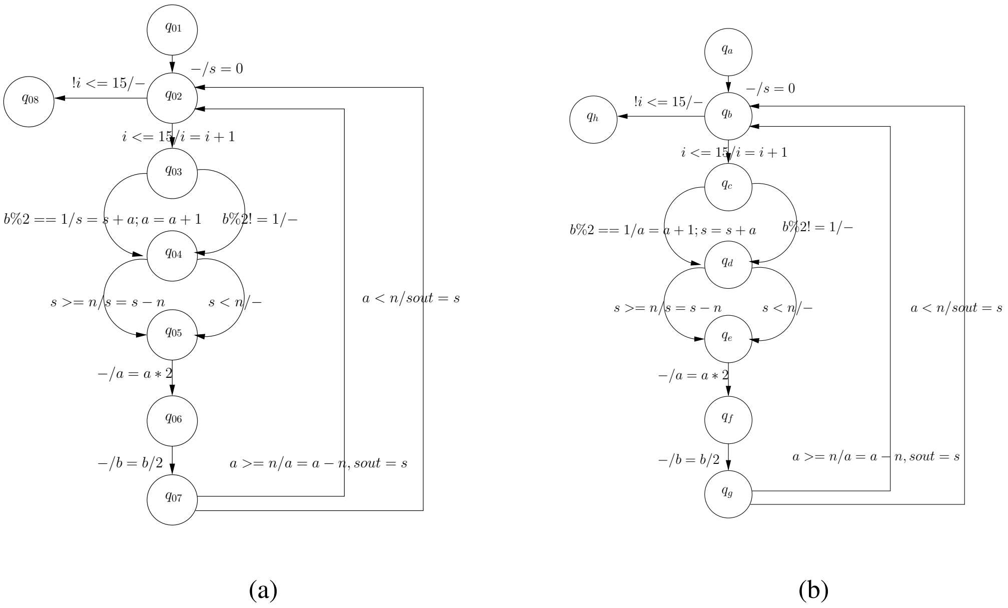
Sub-case 2.2 [$t^g_{\underline{i}}$ has no dependence on any transitions (preceding it)]:

Here the above step obviously attains the above two claims.

The shuffling step S is repeated over all the indexes of p_g where mismatch occurs at every stage retaining the equivalences depicted in claims C1 and C2. So, we have the finally modified path $p_s \cong p^m_s \cong p_g$. (Contradiction) ■

Reporting dependency violation: an example with FSMDs

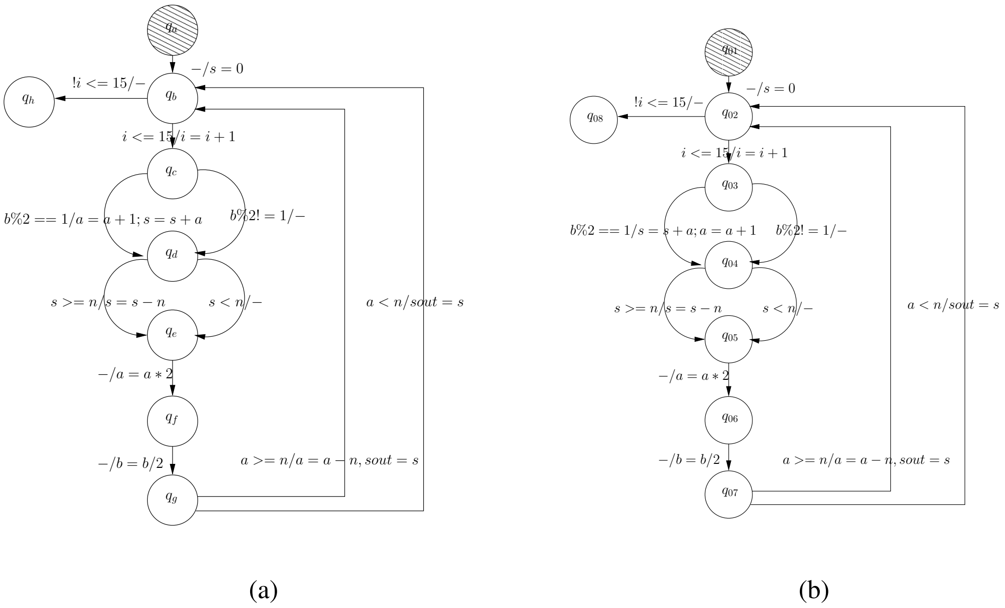
Dependency is violated in the following program segments. In the first one, which is a correct program, the expression a = a + 1 follows the expression s = s + a. In s = s + a, the variable a is dependent on the variable used for storing the input n. In the second program segment, i.e., in the incorrect version, a = a + 1 precedes s = s + a. Thus, the variable a in s = s + a is dependent on a = a + 1. This causes the violation of dependency in the incorrect program. The corresponding FSMDs are given in the figure 2.22 below.



(a) (b)

Figure 2.22: Example of dependency violation.

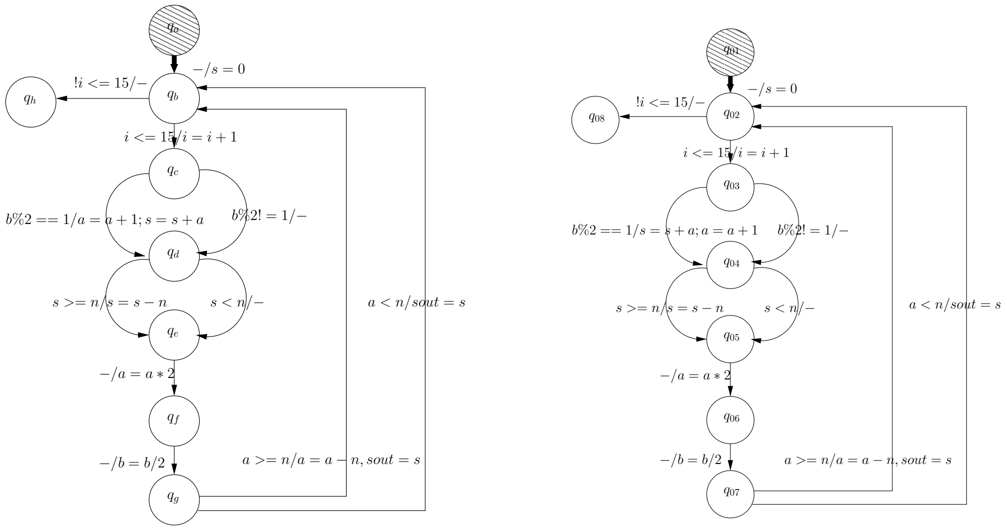
When these two FSMDs are subjected to equivalence checking using the equivalence checker, the latter reports that the two FSMDs are not equivalent, reporting the path for which it was not able to find equivalence. The starting state of the failed path in the incorrect program is the last correct state (LCS). The working of equivalence checker on the two FSMDs is shown in the following figures 2.23 to 2.35.



(a)

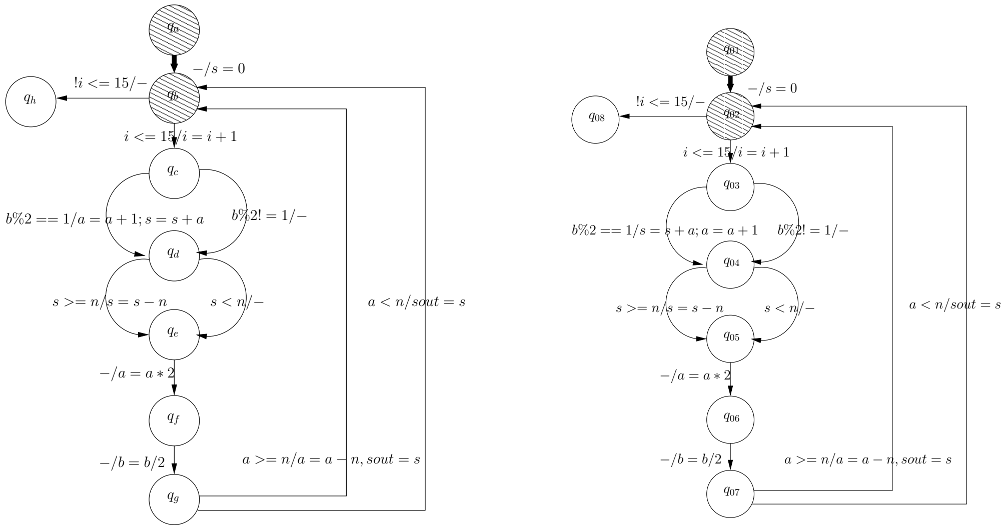
(b)

Figure 2.23: Example of dependency violation: Working of equivalence checker I.



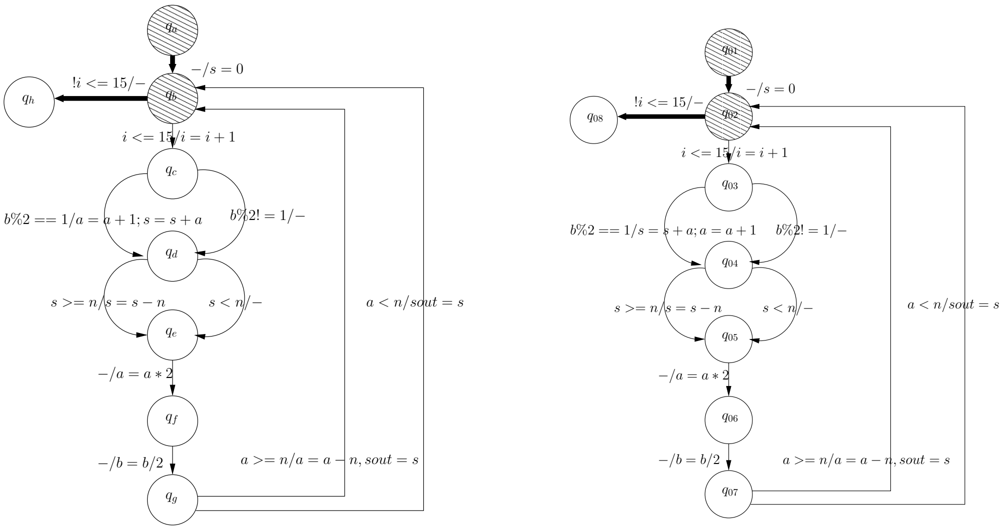
(a) (b)

Figure 2.24: Example of dependency violation: Working of equivalence checker II.



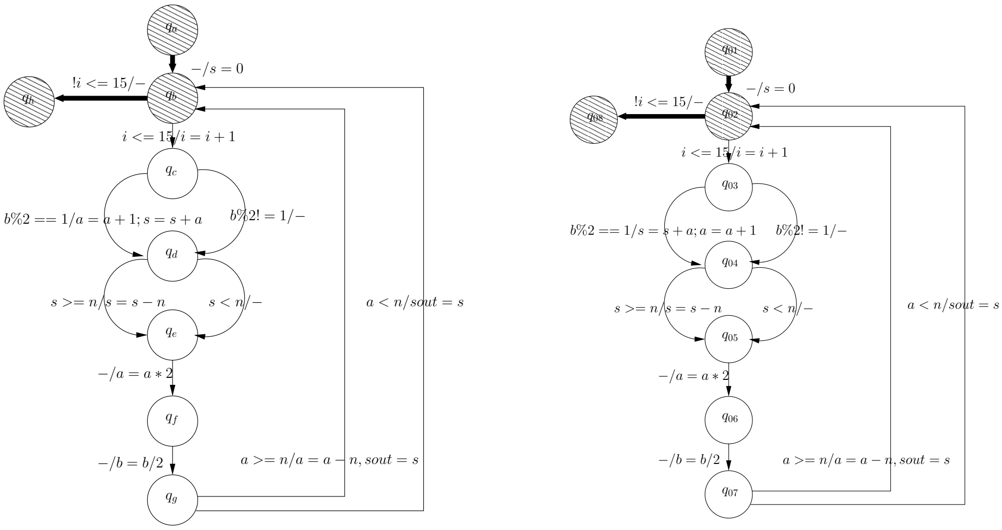
(a) (b)

Figure 2.25: Example of dependency violation: Working of equivalence checker III.



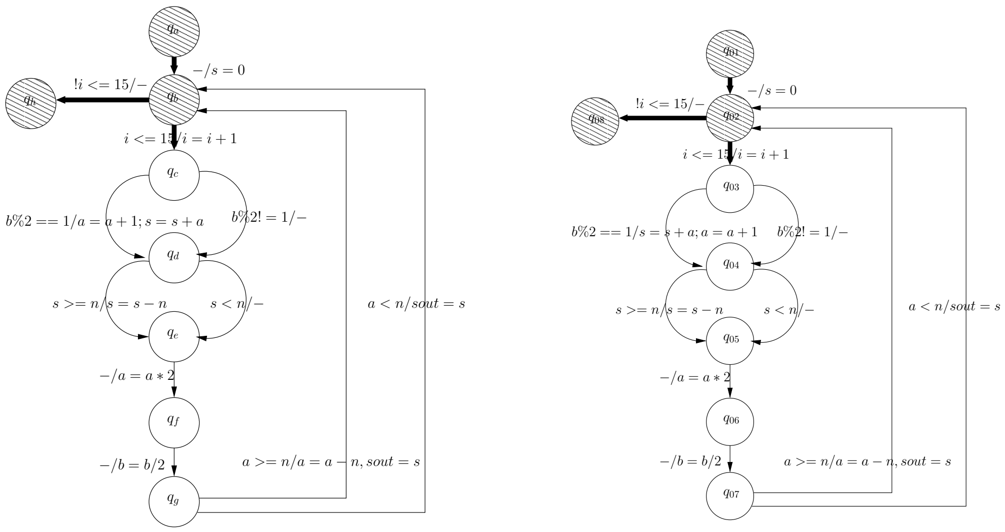
(a) (b)

Figure 2.26: Example of dependency violation: Working of equivalence checker IV.



(a) (b)

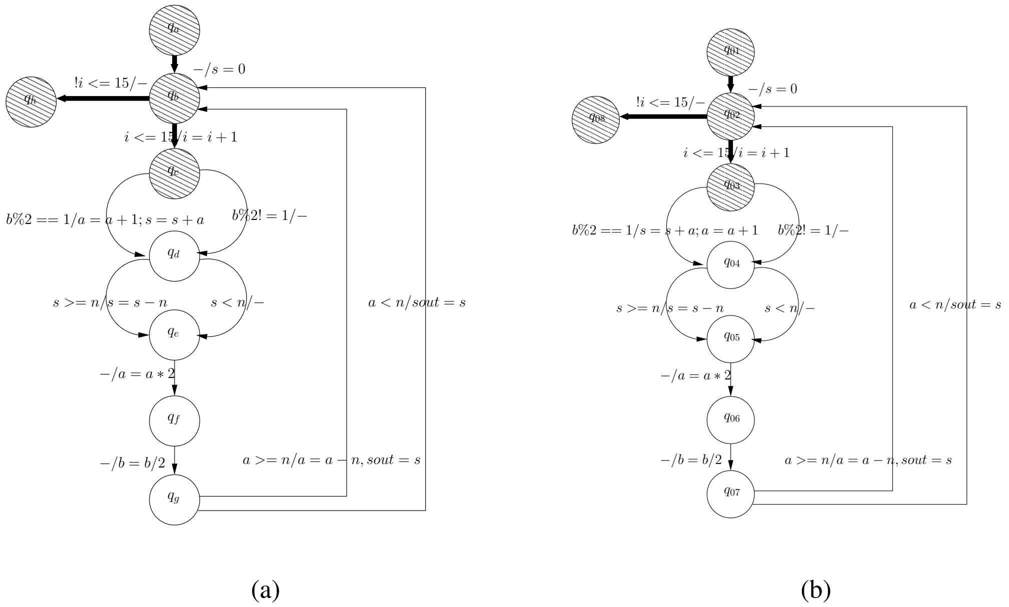
Figure 2.27: Example of dependency violation: Working of equivalence checker V.



(a) (b)

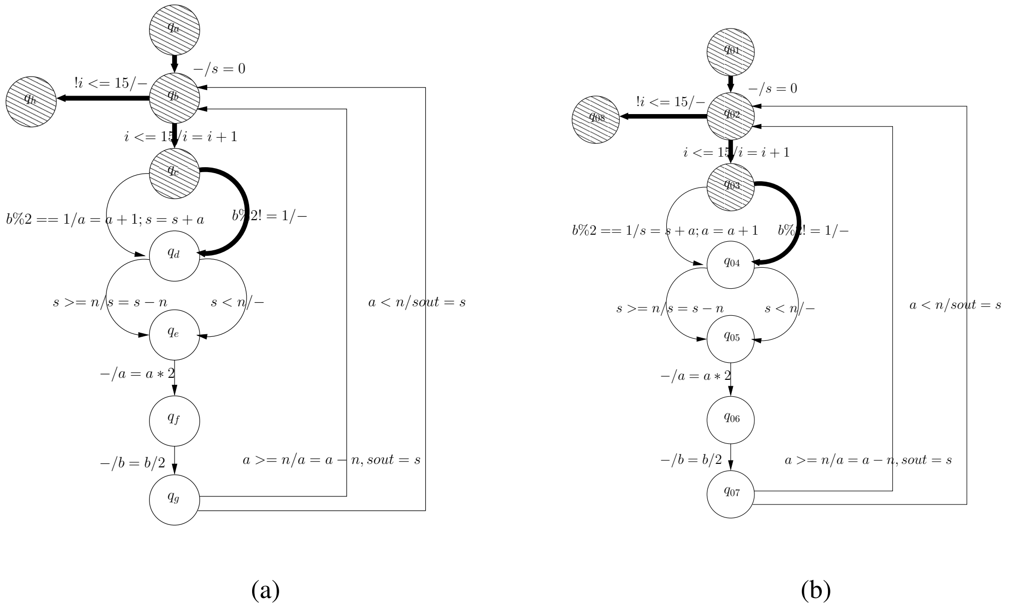
Figure 2.28: Example of dependency violation: Working of equivalence checker VI.

Now when it tries to find equivalent path for the edge shown by dotted line in figure 2.32(a) and it does not find, it extends that path. Some of the steps showing depth first search and backtracking are shown in the following figures 2.32(b) to 2.35. Ultimately the equivalence checker fails.



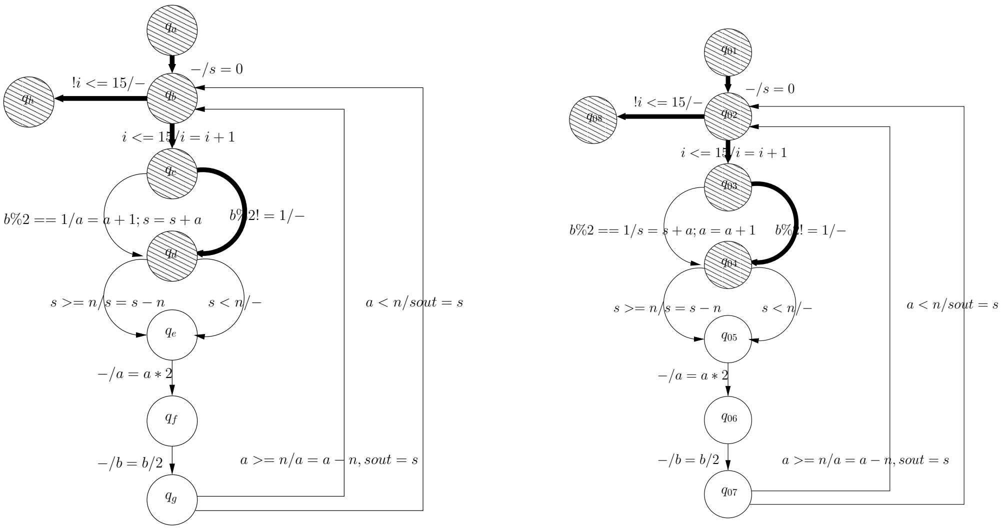
(a) (b)

Figure 2.29: Example of dependency violation: Working of equivalence checker VII.



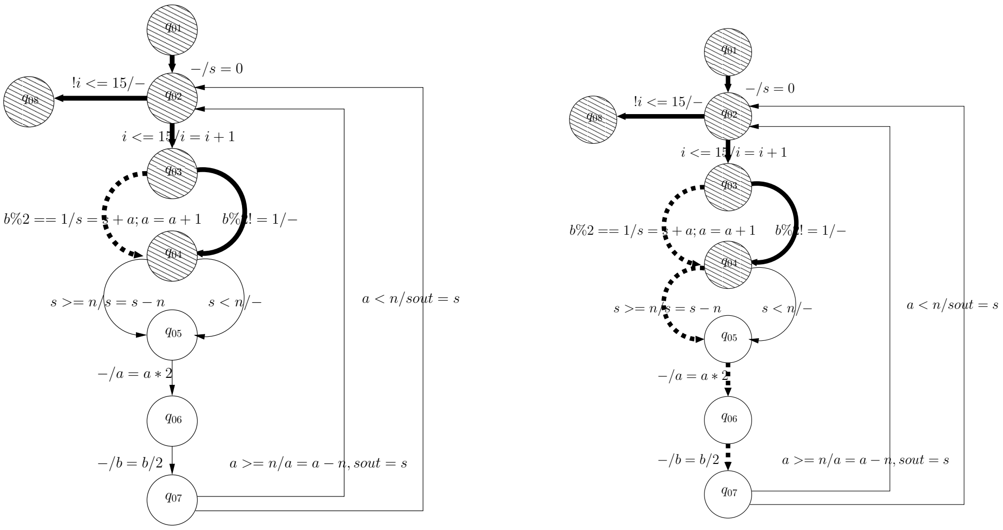
(a) (b)

Figure 2.30: Example of dependency violation: Working of equivalence checker VIII.



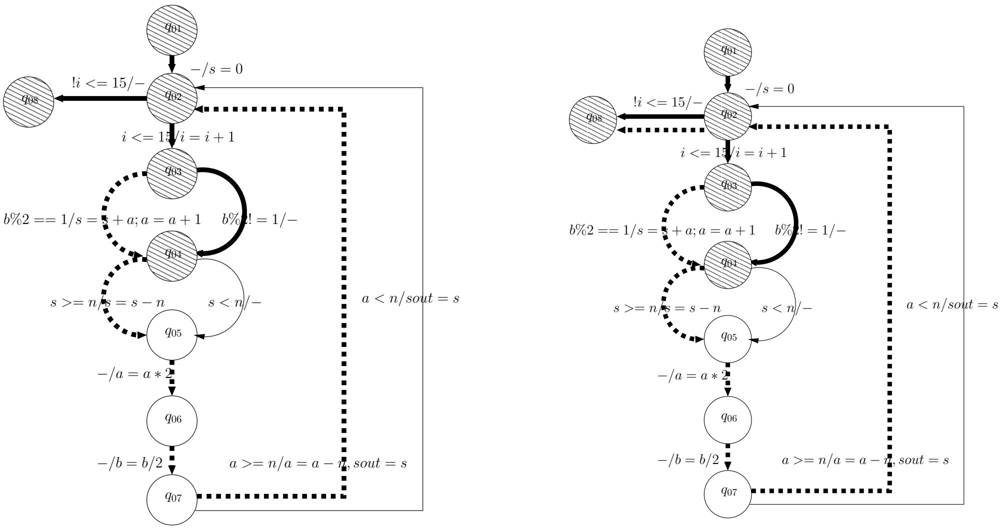
(a) (b)

Figure 2.31: Example of dependency violation: Working of equivalence checker IX.



(a) (b)

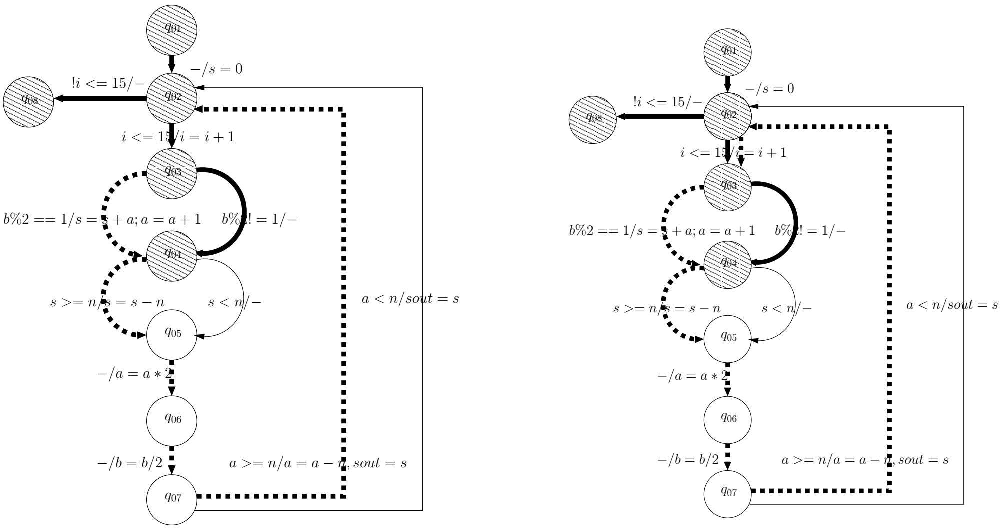
Figure 2.32: Example of dependency violation: Working of equivalence checker X.



(a)

(b)

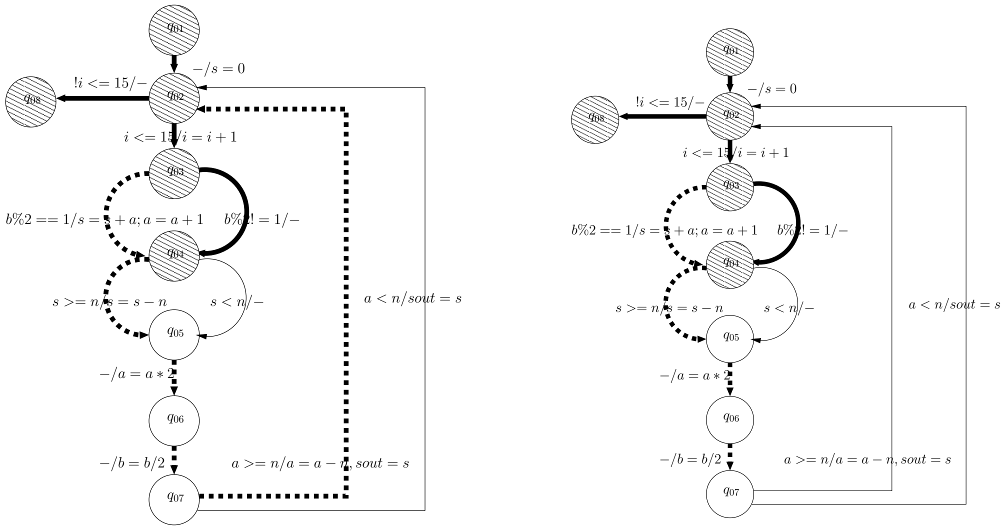
Figure 2.33: Example of dependency violation: Working of equivalence checker XI.



(a)

(b)

Figure 2.34: Example of dependency violation: Working of equivalence checker XII.



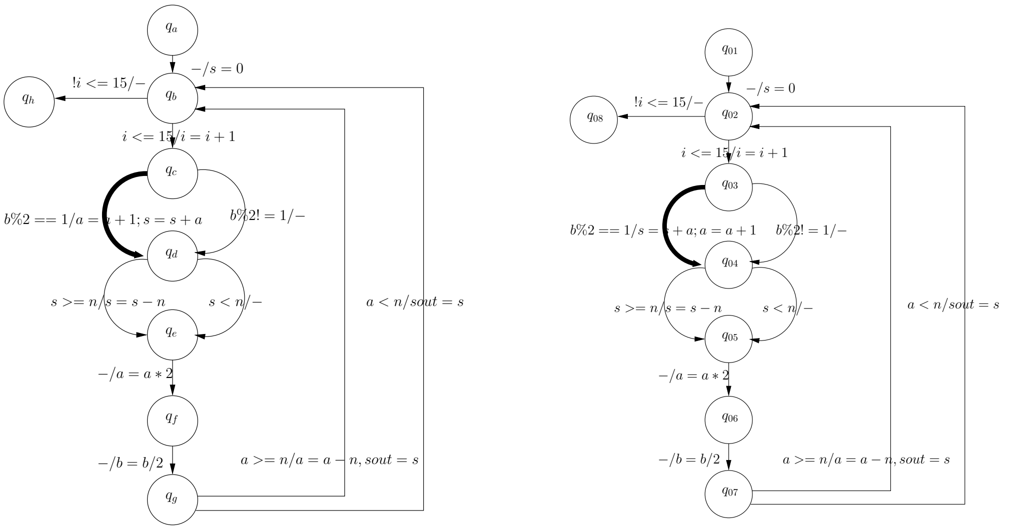
(a) (b)

Figure 2.35: Example of dependency violation: Working of equivalence checker XIII.

While checking containment, all the corresponding paths between the LCS and next cut-point are found unordered path-wise both way contained. The run of the containment checker identifying the unordered path-wise both way contained paths is shown in the following figures 2.36 and 2.37. It is to be noted that the equivalence checker reports non-equivalence for a path starting at the LCS. The respective unordered path-wise both way contained paths on the FSMD of student's program and their containing paths on the FSMD of golden program are shown with the help of dark lines in figures 2.36 and 2.37 in figures (b) and (a) respectively. The algorithm that has been used to detect dependency violation, is given later in this section. In the following we discuss some preliminaries related with the detection algorithm for dependency violation.

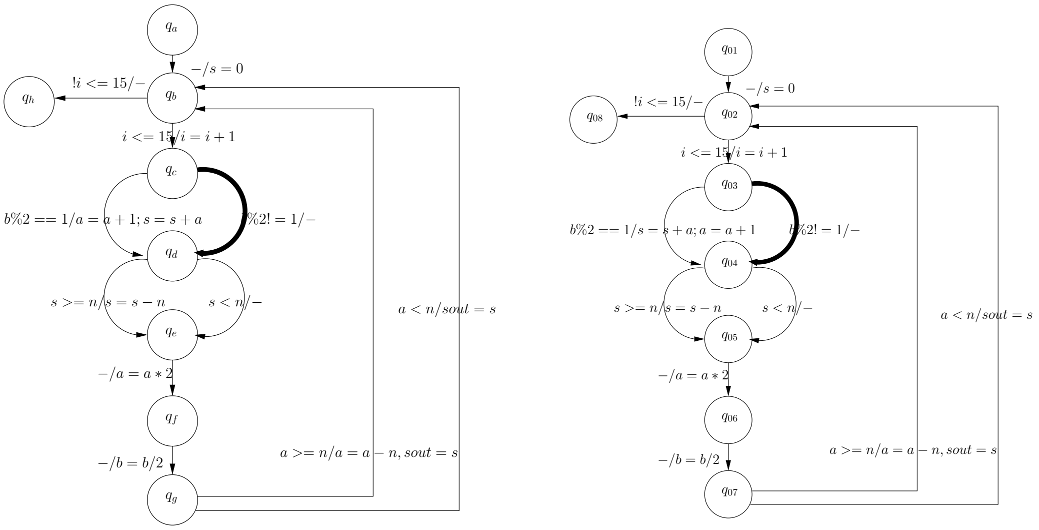
Dependency graph

A dependency graph is a directed partially ordered graph, in which along-with the variable being defined, the statement number also appears at the nodes. Edges represent the dependency relation between the statements at the two ends of the edge. We introduce an edge from node for statement numbered 1 to node for statement numbered 2, if the statement numbered 2 is dependent on statement numbered 1. The idea of dependency graph is explained with the program codes shown below and the



(a) (b)

Figure 2.36: Example of dependency violation: Working of containment checker showing the unordered path-wise both way contained paths I.



(a) (b)

Figure 2.37: Example of dependency violation: Working of containment checker showing the unordered path-wise both way contained paths II.

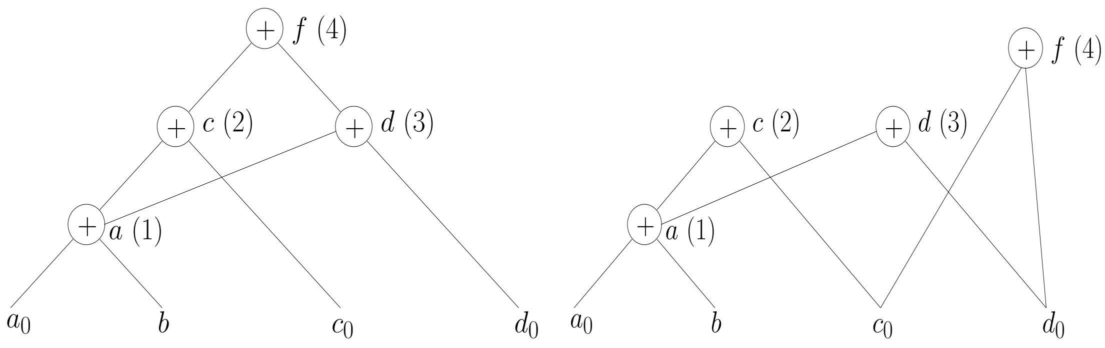
corresponding directed acyclic graphs (DAG) of the golden and student's programs in figure 2.38 below, built as per the algorithm 9.2 for constructing a DAG described in Aho et al. [6]. The numbers in brackets on the node labels in figure 2.38 are the statement numbers in the golden program, where the variable shown on the node label is defined. These numbers, which are from the golden program are used for constructing both the DAGs in figure 2.38.

Listing 2.3: Golden program

```
void code()
{


   1: a = a + b;
   2: c = c + a;
   3: d = d + a;
   4: f = c + d;


}
```

Listing 2.4: Student's program

```
void code()
{


   1(4): f = c + d;
   2(1): a = a + b;
   3(2): c = c + a;
   4(3): d = d + a;


}
```



(a) (b)

Figure 2.38: DAGs of (a) golden program and (b) student's program.

The dependency graphs of figures 2.39 and 2.40 may then be visualized from the DAG representation of the golden and the student's programs shown in figure 2.38.

The dependencies may also be depicted in a dependency graph derived from a DAG as in Figure 2.38; an edge is present there, if a dependency is also present in the DAG. In the dependency graph an edge is present between two nodes, *m* and *n*, if statement numbered *n* depends on the statement numbered *m* in the DAG. The dependencies in golden program are, $1 \prec 2$, $1 \prec 3$, $3 \prec 4$ and $2 \prec 4$. The dependencies

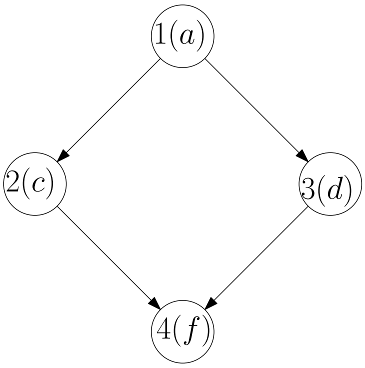


Figure 2.39: Dependency graph of golden program

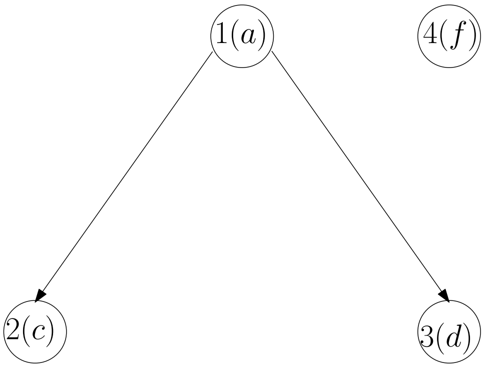


Figure 2.40: Dependency graph of student's program

in student's program are, $1 \prec 2$ and $1 \prec 3$.

Accordingly, figures 2.39 and 2.40 show the edges representing dependencies, e.g., in figure 2.39, the statements with numbers 2 and 3 are dependent on the statement numbered 1, the statement numbered 4 depends on the statements numbered 2 and 3. The figure 2.40 has the node numbered 4 alone, as no node depends on it.

Dependency violation is detected by comparing the dependency graphs in the two programs, by picking up each edge in the dependency graph of golden program by traversing it in a depth first manner and searching for it in the dependency graph of student's program. In case an edge is not found in the dependency graph of the student's program, then the two statements at the two ends of the edge are concluded to violate dependency and both the violating statements (pairwise) are reported to the student.

If the containment checker resorts to path extension, then basic blocks along the extended path must be concatenated to obtain extended basic block for both the paths in M_g and M_s and then those can be compared in a similar manner. It may be noted that the path extension never goes beyond the loop entry point.

As the example for locating the occurrence of dependency violation, we once again consider figures 2.39 and 2.40. We first select the edge $1 \rightarrow 2$ in figure 2.39, and try to find whether it exists in figure 2.40, applying depth first search (DFS). We find that the edge $1 \rightarrow 2$ is present in figure 2.40. It can, therefore, be concluded that the dependency $1 \prec 2$ is preserved in the student's program. The next edge is $2 \rightarrow 4$, which is searched for in figure 2.40. As there is no outgoing edge from node 2, it is concluded that the edge 2 → 4 is not present. There is, thus, a dependency violation between statements 2 and 4 in the student's program, which is reported to the student. As we are searching for the edges in figure 2.39, and we have come to node 4, from which there is no other outgoing edge, we backtrack up to the node 1, from where we started. We then pick up the other outgoing edge from node 1, i.e., the edge 1 → 3. We look out for this edge in figure 2.40, starting in a depth first manner from node 1. The edge is found to be present in figure 2.40. The next edge 3 → 4, in figure 2.39, is similarly searched in figure 2.40 and is not found there. The dependency violation between statements 3 and 4 is thus located in the student's program and is reported to the student.

Implementation: In the implementation, the dependencies of statements which are depicted in the DAG are represented as individual singly linked lists per statement as given in the figures 2.1 and 2.2. The partial order captured by the DAG representation is preserved in the singly linked list representation.

A dependency list here is a singly linked list of statements on which a particular statement depends. The first node of the dependency list contains the statement number and subsequent nodes contain the number of the statements on which there is a dependence. This is explained with a code and its dependency list below.

How it is made?

The making of dependency list for each statement s, extracted from the FSMD, involves the following steps. For each variable v occurring on the rhs in s, if l_s be the last statement where v is defined (if such a statement exists) then s depends on l_s. Hence we add node l_s to the dependency list of s.

We now define containment equivalence of paths, which has been used later in the description of method of reporting dependency violation

Listing 2.5: Golden program

```
void code()
{


   1: a = a + b;
   2: c = c + a;
   3: d = a + f + c;


}
```

Listing 2.6: Student's program

```
void code()
{


   1(2): c = c + a;
   2(1): a = a + b;
   3(3): d = a + f + c;


}
```

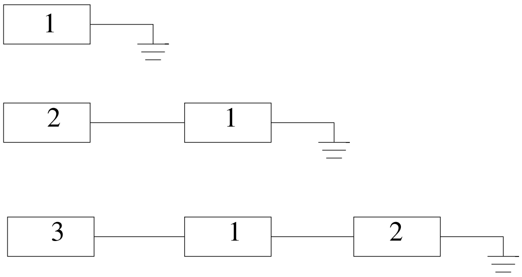


Figure 2.41: Dependency list of golden program

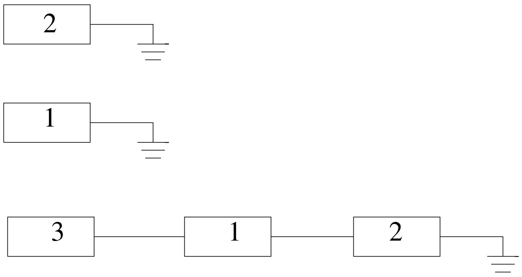


Figure 2.42: Dependency list of student's program

For reporting dependency violation, we compare the dependency lists for all the statements in the FSMD M_g and the FSMD M_s, which have been found to be *containment equivalent* by the containment checker and the un-matching lists are used to report dependency violation. The dependency information in the two un-matching lists is reported as dependency violation.

The algorithm for detecting dependency violation is given in algorithm 2. The details of the steps in the algorithm are as follows.

1. In step 1, equivalence checker is executed, followed by containment checker on failure of equivalence checker and the output of containment checker is checked in step 2. If the containment checker reports that all the paths between LCS and the next cut-point are unordered path-wise both way contained then step 3 is executed.

2. Details of step 3 are as follows. In this step a dependency list is made. Dependency list has the dependency information of a statement in the path in terms

of previous statements. Individual statements are extracted from the FSMDs, which lie on each path p_s in M_s given in step 2 and it's containment equivalent path p_g in M_g. Then statement numbers are annotated to the statements extracted. A singly linked list of nodes with statement numbers of the previous statements on which the statement s depends is made as follows. For each extracted statement s, corresponding to every variable v occurring on the rhs in s, if l_s be the last statement where v is defined (if such a statement exists), then s depends on l_s. A node l_s is added to the dependency list of s.

3. Details of step 4 are as follows. In this step, the dependency lists of the two programs are compared for each statement s obtained from the FSMD of golden program in step 2. If dependency violation is detected on comparing the dependency lists for the statement s in the two FSMDs, it is reported as follows. Let the dependency lists for the statement s in the two FSMDs be $lists_g$ and $lists_s$. Let $lists_g = \{s_1, s_2, \ldots, s_k\}$, $lists_s = \{s'_1, s'_2, \ldots, s'_{k'}\}$. If $\exists s_i \notin \{s'_1, s'_2, \ldots, s'_{k'}\}, 1 \leq i \leq k'$ OR $\exists s'_i \notin \{s_1, s_2, \ldots, s_k\}, 1 \leq i \leq k$, then the two lists $lists_g$ and $lists_s$ are reported.

Algorithm 2: Detect_dependency_violation

```
    Input: $M_s$, FSMD of student's program and $M_g$, FSMD of teacher's program;
    Output: Report dependency violation, if any;
L1  Begin;
    /*Step 1:  Execute equivalence checker followed by containment checker on failure          */
L2  execute the equivalence checker with the following parameters: (i)$M_s$ and (ii)$M_g$
L3  if equivalence checker reports $M_s \not\sqsubseteq M_g$ then
L4      execute the containment checker with the following parameters: (i)$M_s$ and (ii)$M_g$
    /*Step 2:  Check the output of containment checker                                          */
L5  if the containment checker reports that all the paths between LCS and the next cut-point are unordered path-wise both
      way contained then
L6      go to L9, step 3
L7  else
L8      exit the dependency violation checker
    /*Step 3:  Make dependency list.  Dependency list has the dependency information of a
      statement in the path in terms of previous statements.                                   */
L9  for each path $p_s$ in $M_s$ given in step 2 and it's containment equivalent path $p_g$ in $M_g$ do
L10     extract individual statements from the path
L11     annotate statement numbers to the statements extracted
        /*make a singly linked list of nodes with statement numbers of the previous statements on
          which the statement $s$ depends                                                      */
L12     for each statement s extracted do
L13         for each variable v occurring on the rhs in s do
L14             if $l_s$ be the last statement where v is defined (if such a statement exists) then
                    /*$s$ depends on $l_s$                                                     */
L15                 add node $l_s$ to the dependency list of $s$
    /*Step 4:  Compare the dependency lists of the two programs.  If dependency violation is
      detected, report it and exit                                                             */
L16 for each statement s obtained from the FSMD of golden program in the previous step do
L17     Compare the dependency lists $lists_g$ and $lists_s$ for the statement $s$ in the two FSMDs
L18     Let $lists_g = \{s_1, s_2, \ldots, s_k\}$, $lists_s = \{s'_1, s'_2, \ldots, s'_{k'}\}$
L19     if $\exists s_i \notin \{s'_1, s'_2, \ldots, s'_{k'}\}, 1 \leq i \leq k'$ OR $\exists s'_i \notin \{s_1, s_2, \ldots, s_k\}, 1 \leq i \leq k$ then
L20         report $lists_g, lists_s$
L21 End
```

2.3.2 Reporting errors of parenthesis skipping

The case of error in the student's program when the closing parenthesis is wrongly put beyond its proper place in the code by the student causes parenthesis skipping error. For example, if the closing parenthesis of the first if-else block has been put at the end of the second if-else block in a program where two consecutive if-else blocks are present. An error like this in the code causes one of the transitions emanating from the starting state of first if-else block, to skip meeting the final state of this block and instead becomes incident on the final state of the next if-else block. We refer to this type of error as that of *skipping parenthesis*. In this case the containment checker reports *unordered path-wise both way contained and path-wise one way contained for faulty branching*. As an example of parenthesis skipping error, consider the following code. The corresponding FSMDs are shown in Figures 2.1 and 2.2 respectively. In the FSMD of student's program in Figure 2.1, the edge from the state q_c skips meeting at state q_d and instead meets the next state q_e due to wrong placement of closing parenthesis of the first if block.

Listing 2.7: Golden program

```
void main()
{
  int s = 0, i = 0;
  while (i <= 15){
     i = i + 1;
     if (b % 2 == 1){
        s = s + a;
     }
     if (s >= n){
        s = s - n;
     }
     b = b / 2;
     a = a * 2;
     if (a >= n){
        a = a - n;
     }
  }
  sout = s;
}
```

Listing 2.8: Student's program

```
void main()
{
  int s = 0, i = 0;
  while (i <= 15){
     i = i + 1;
     if (b % 2 == 1){
        s = s + a;
        if (s >= n){
           s = s - n;
        }
     }
     b = b / 2;
     a = a * 2;
     if (a >= n){
        a = a - n;
     }
  }
  sout = s;
}
```

From the angle of our analysis, the parenthesis skipping may be of two types.

1. Skipping parenthesis does not belong to any loop.

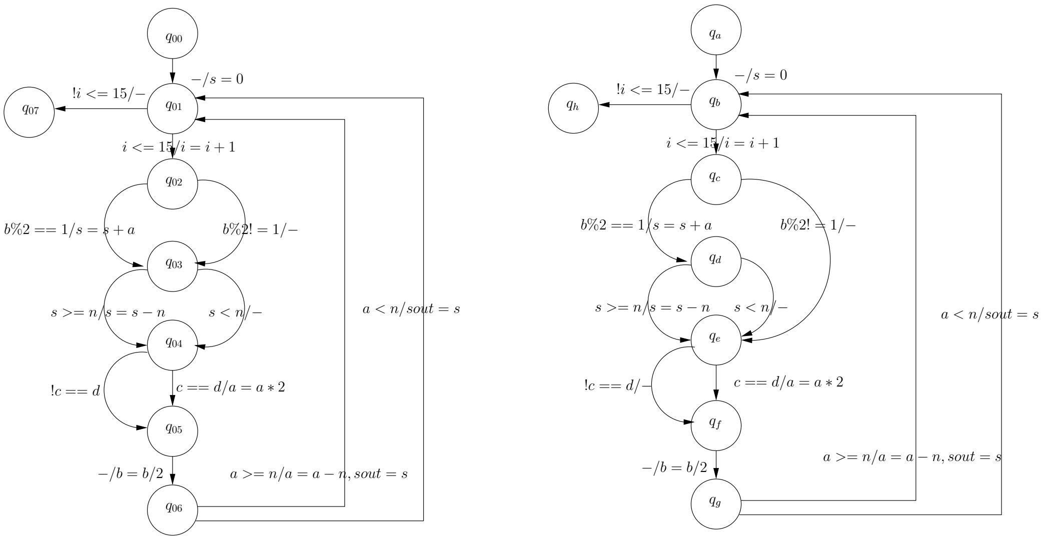
2. Skipping parenthesis belongs to a loop.

1. Skipping parenthesis does not belong to any loop

There are two possibilities in this type of error.

Case I: The edge, which is skipping parenthesis, meets a cut-point.

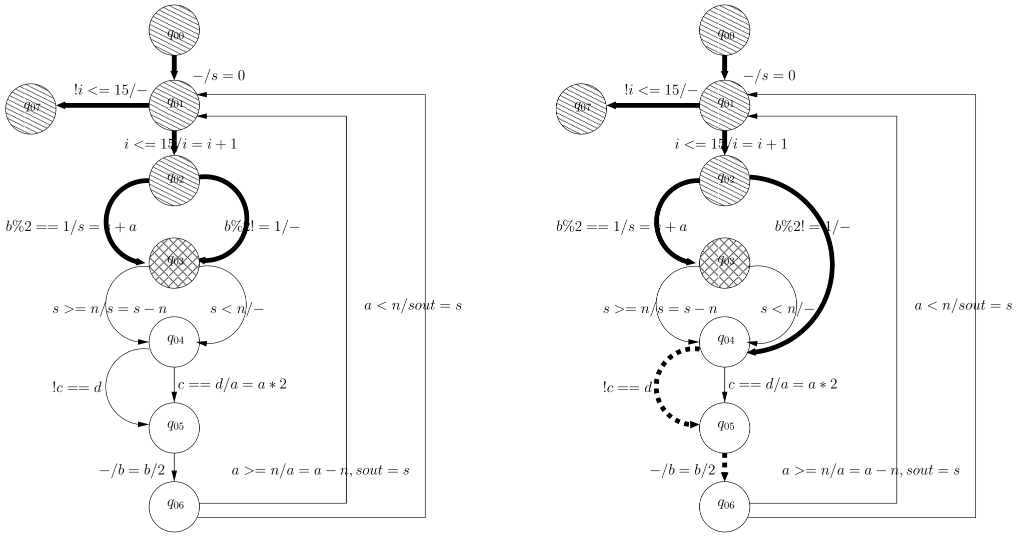
In this case, the cut-point where the skipping edge meets, is reported as the LCS. The containment checker will report that the incorrect path starting from the LCS to the next cut-point is path-wise one way contained in the correct FSMD. However, this fact has not been exploited in the algorithm for this case, as this portion of incorrect FSMD does not contain any error. The error lies in incorrect FSMD above the LCS, which is handled by the algorithm for this case. Actually, the error lies between the LCS and the least numbered state, from which transition is coming to LCS; as this transition corresponds to the maximum possible portion of the erroneous code, which includes the skipping of parenthesis error. This case is explained with the help of following simulation for the FSMDs shown in figure 2.43.



(a) (b)

Figure 2.43: Case I: Figures showing (a) M_g and (b) M_s

In the execution of equivalence checker shown in figure 2.44(a), the dark edges and states indicate that the execution of equivalence checker has found their equivalence in the incorrect FSMD. The figure 2.44(b) shows the path with the help of dark dotted line, for which equivalence checker fails to find an equivalence. Also the dark edges and states in this figure are the corresponding equivalents of their counterparts in the correct FSMD in figure 2.44(a).



(a) (b)

Figure 2.44: Case I: Figures showing equivalent states in (a) M_g and (b) M_s

Case II: The edge which is skipping parenthesis does not meet a cut-point.

In this case, the edge which is skipping parenthesis, meets a state which is not a cut-point. The cut-point state where the skipping edge started is reported as the LCS. The containment checker will report that the incorrect path, starting from the LCS to the next cut-point, is path-wise one way contained in the correct FSMD. This case is explained with the help of the following simulation of the equivalence checker on the FSMDs in figure 2.45.

In the following figure 2.46, the path shown by dotted line, is the path for which equivalence checker fails to find equivalence. Also the dark edges and states in this figure are the corresponding equivalents of their counterpart in the correct FSMD of figure 2.45(a).

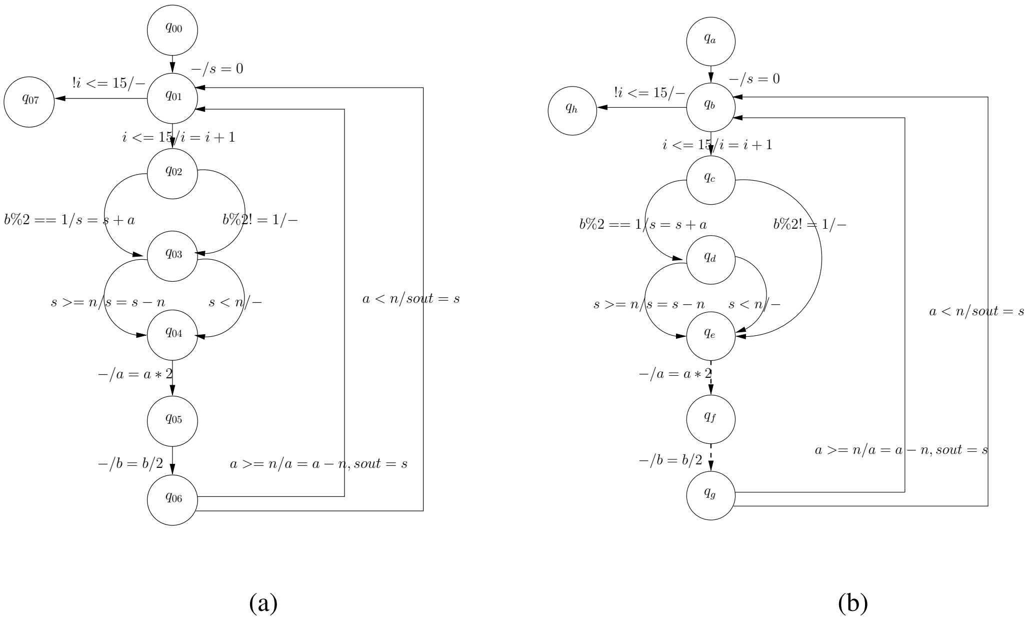


Figure 2.45: Case II: (a) M_g and (b) M_s

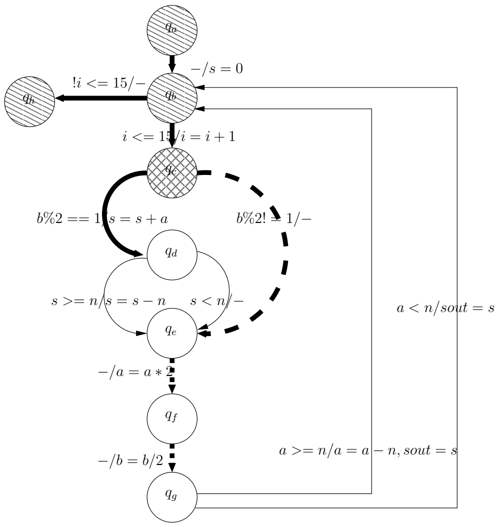


Figure 2.46: Case II: Equivalence found for the dark edges but not for the dotted edges of M_s

Algorithm for case I

The state in correct FSMD, reported to be corresponding to the LCS by the equivalence checking mechanism, may not be the actual corresponding state, due to the error of skipped parenthesis. Under this error, the edge corresponding to skipped parenthesis may possibly be incident on the LCS. To detect the possibility of such an error, if any, the following algorithm is used. This algorithm is capable of detecting the error in question, if there were an error. If the error of skipping parenthesis is not there, the execution of the algorithm will not come in the way of detecting further errors. Hence in both the cases of error, whether there is skipping of parenthesis or not, the following algorithm 3 is executed.

Algorithm 3: Skipping_Edge_Meets_a_Cut_Point

```
Input: $M_s$, FSMD of student's program and $M_g$, FSMD of teacher's program;
Output: Missing or extra incoming edge on states starting from the least
        numbered state in FSMD of student's incorrect program;
Begin;
/*Step 1:  Identify the least numbered state.  The state
  number is given by equivalence checker in the order in which
  they are created.                                          */
Find the least numbered state among the states from which the transitions are
  coming to the LCS;
/*Step 2:  Find the corresponding state of the least numbered
  state                                                      */
Find from the correct FSMD the state corresponding to the least numbered state
  found in the above step;
/*Step 3:  Run the algorithm to find out the missing or extra
  incoming edge on the states starting from the least numbered
  state in incorrect FSMD and exit there after.              */
Call the algorithm to find out the missing or extra incoming edge on the states
  starting from the least numbered state in incorrect FSMD and exit there after.
  The algorithm has been given below.
End
```

Main steps

1. Find the least numbered state among the states from which the transitions are coming to the LCS. The state number is given by equivalence checker in the order in which they are created.

2. Find from the correct FSMD the state corresponding to the least numbered state found in the above step.

3. Run the algorithm to find out the missing or extra incoming edge on the states starting from the least numbered state in incorrect FSMD and exit there after. The algorithm has been given below.

Algorithm 4: Finding_missing_or_extra_incoming_edge

```
    Input: (i) State from the FSMD of incorrect student's program, from where the algorithm should find out the missing
           or extra incoming edge, (ii) its corresponding state on the other FSMD, (iii) $M_s$, the FSMD of student's program
           and (iv) $M_g$, the FSMD of teacher's program;
    Output: The incorrect code segment and the corresponding correct code segment;

    /*This is an algorithm to find out the missing or extra incoming edge incident on the states
      starting from given state in incorrect FSMD                                              */
L1  Begin;
    /*Step 1:  Create a list of incoming edges for each state                                 */
L2  for each FSMD do
L3      for each state do
L4          create a linked list of incoming edges

    /*Step 2a:  initial push on stack1                                                        */
L5  Find out the set difference of incoming edges on the given state (in the incorrect FSMD) and its corresponding state (in
      the correct FSMD).
L6  Push all the edges in the set difference on stack1.
    /*Step 2b:  initial push on stack2                                                        */
L7  Find out the set difference of incoming edges on the corresponding state (in the correct FSMD of the given state) and
      those on the given state (in the incorrect FSMD).
L8  Push all the edges in the set difference on stack2.
    /*Step 3:  Popping the stack if their top have the same entry                             */
L9  if the content of both the stacks are not empty and the top of both the stacks are the same then
L10     we pop both the stacks;
L11 else
L12     select minimum numbered state among the states on which transitions from the given state are incident;
L13     goto L5, step 2a with the minimum numbered state as the given state;

    /*Step 4:  Output the Incorrect code and correct code segments                            */
L14 Output the code from incorrect file between the two end point states of the popped out edge from stack2 as incorrect
      portion of the code;
L15 Output the correct portion of code is the code from correct file between the two end point states of the popped out edge
      from stack1;
L16 End
```

Main steps to find out the missing or extra incoming edge incident on the states starting from given state in incorrect FSMD

1. A linked list of incoming edge for each state is created for both the FSMDs.

2. (a) Find out the set difference of incoming edges on the given state (in the incorrect FSMD) and its corresponding state (in the correct FSMD). Push all the edges in the set difference on stack1.

   (b) Find out the set difference of incoming edges on the corresponding state (in the correct FSMD of the given state) and those on the given state (in the incorrect FSMD). Push all the edges in the set difference on stack2.

3. If the content of both the stacks are not empty and the top of both the stacks are the same, then we pop both the stacks, else select minimum numbered state among the states on which transitions from the given state are incident and goto *step 2a* with the minimum numbered state as the given state.

4. Output the code from incorrect file between the two end point states of the popped out edge from stack2 as incorrect portion of the code. The correct portion of code is the code from correct file between the two end point states of the popped out edge from stack1.

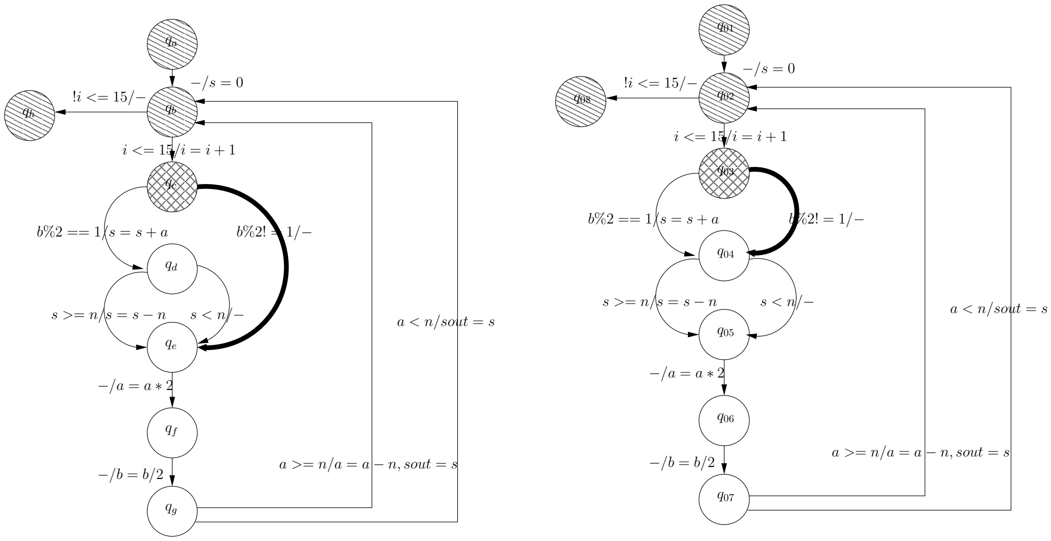
Algorithm for case II

The algorithm for this case is given in algorithm 5. It works as follows. Run the algorithm to find out the missing or extra incoming edge incident on the states starting from the LCS reported by the equivalence checker as the given state in incorrect FSMD.

After the execution of the above algorithm the entry corresponding to the highlighted edges are stored on stack1 and stack2 (figure 2.47).

Algorithm 5: Skipping_Edge_Does_not_Meet_a_Cut_Point

```
Input: $M_s$, FSMD of student's program and $M_g$, FSMD of teacher's program,
       Last Corrected State;
Output: Missing or extra incoming edge on states starting from the least
        numbered state in FSMD of student's incorrect program;
Begin;
/*Step 1:                                                    */
Run the algorithm to find out the missing or extra incoming edge incident on
  the states starting from the LCS reported by the equivalence checker as the
  given state in incorrect FSMD.
/*Step 2:                                                    */
After the execution of the above algorithm the entry corresponding to the
  highlighted edges are stored on stack1 and stack2.
End
```



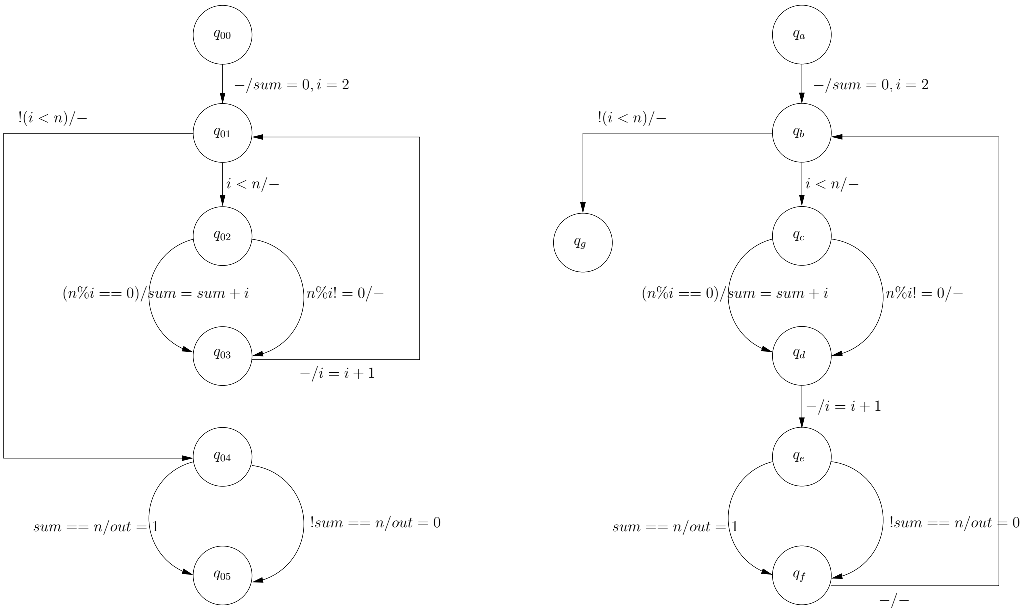
(a)

(b)

Figure 2.47: Figures showing the output of algorithm for case II. Dark edges are the entry present in stack1 and stack2.

2. Skipping parenthesis belongs to a loop

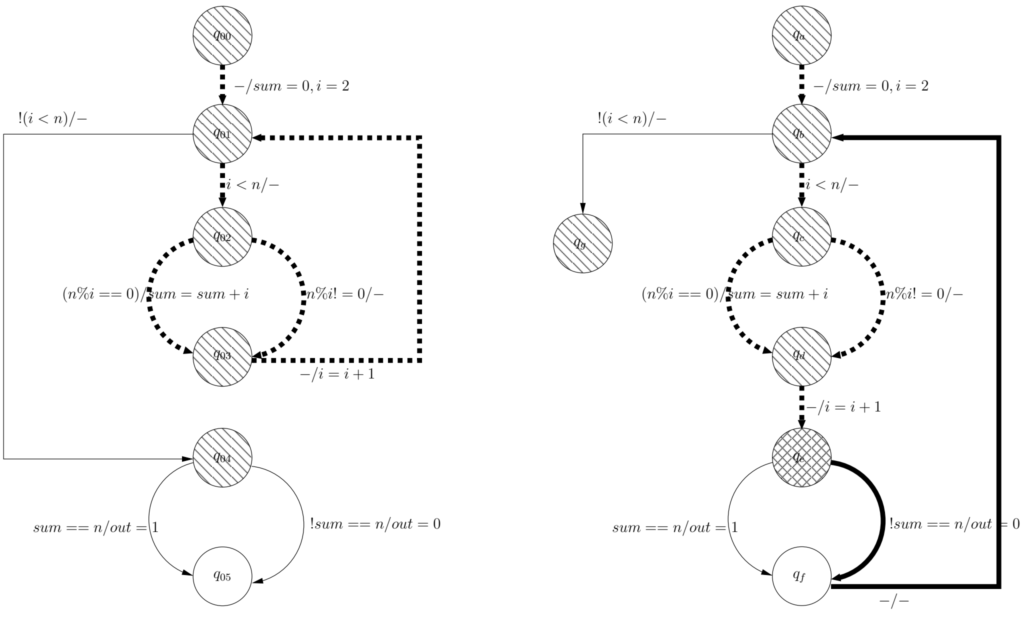
Consider a case of a for loop, where the parenthesis is skipped beyond the for loop, in incorrect code. As a result, the code that was outside the for loop in the correct code, becomes a part of the for loop in the incorrect code. Our job is to find out this extra piece of code inside the loop and move it outside the loop. Because of the skipped parenthesis and the resulting introduction of extra code inside the loop body, the start state of the for loop in the correct FSMD, has got two corresponding states in the incorrect FSMD, one of which is obviously the start state of the for loop in the incorrect FSMD. The other corresponding state is the LCS.
The FSMDs in the following figure 2.48 show an example of above discussed case.



(a) (b)

Figure 2.48: Figures showing (a) M_g and (b) extension of loop in M_s due to skipping parenthesis.

In the above figure 2.48, the simulation of equivalence checker results in the following dotted edges and darkened states in figure 2.49, showing that the execution of equivalence checker has found their equivalence / correspondence in the incorrect FSMD. The figure 2.49(b) on the right hand side shows the path with the help of a dark line, for which the equivalence checker fails to find an equivalence. Also, the dotted edges and darkened states in this figure are the equivalents / correspond to their counterparts in the correct FSMD of figure 2.49(a).



(a) (b)

Figure 2.49: Figures showing (a) M_g and (b) M_s, showing equivalence checker failing for path shown with dark line

Algorithm for moving the extra code away from the loop

The paths between the LCS and the start state of the loop is reported contained in the correct FSMD by the containment checker. Suggest the code corresponding to the paths between the LCS and the start state of the loop as extra code in the loop; this code should be moved away from the loop.

2.3.3 Error of missing block of code

In this case as the two FSMDs will not be equivalent, we verify this fact by checking the equivalence in both the directions, as checking equivalence only in one direction may sometimes give misleading result about equivalence. When equivalence is checked in both the directions, then non-equivalence will be reported in at least one direction. The LCS is obtained from that run of equivalence checker, among the two runs as suggested above, in which non-equivalence was reported. In case of missing block of code, the containment checking may give two types of results. Both of which are treated in the same way by the algorithm. Discussion of the three cases of missing block of code, two cases in which code block is missing in student's program and one case of block of code found missing in golden program as student's program may have some extra block of code (which has to be removed subsequently as golden program is considered to be correct and anything extra in student's program has therefore to be discarded), is given below.

1. The first case is when the equivalence checker reports computational containment of the FSMD of student's program with the FSMD of golden program. However, as the two FSMDs are not equivalent, so the equivalence checker is executed again, in the other direction, to find the computational containment of the FSMD of golden program with the FSMD of student's program. This time non-computational containment will be reported and so will be the LCS. Now if the containment checker is executed to find the containment of all paths from the reported LCS to the next cut-point of the FSMD of golden program inside the FSMD of student's program and it reports **path-wise un-contained**, then this would mean that there is some code missing in the FSMD of student's program. Moreover, the missing portion of the code is at least the code between the reported LCS and the next cut-point. However, this reported LCS is situated in the FSMD of golden program. By our convention LCS should be from the FSMD of student's program. So, we take the corresponding state (in the FSMD of student's program) of the reported LCS as the LCS. The algorithm executed after this step is given later in this section.

2. In the second case, the equivalence checker reports non-computational contain-

ment of the FSMD of student's program with the FSMD of golden program, along with the LCS. The containment checker reports the path between the LCS and the next cut-point of the FSMD of student's program as **path-wise one way contained** in the FSMD of golden program. As the equivalence checker reports equivalence of the FSMDs from the starting state up to the LCS in the FSMD of student's program and the corresponding state of LCS in the FSMD of golden program; moreover, as the code after LCS is found contained in the FSMD of golden program, this means that there is some code extra in the FSMD of golden program which precedes the code that was found contained in the FSMD of golden program. This extra code of the FSMD of golden program lies between CSLCS, the state that is corresponding to LCS and the state on the FSMD of golden program where the contained code begins. The extra code, therefore, needs to be introduced in the FSMD of student's program. The mechanism for which is same as the mechanism for the first case, after removing extra code, if any, after LCS in M_s.

3. In the case where the student's program has some extra code, which needs to be removed, the equivalence checker reports non-computational containment of the FSMD of student's program with the FSMD of golden program, along with the LCS. The containment checker reports the path between the LCS and the next cut-point of the FSMD of student's program as **path-wise un-contained** in the FSMD of golden program. As the equivalence checker reports equivalence of the FSMDs from the starting state up to the LCS in the FSMD of student's program and the corresponding state of LCS in the FSMD of golden program; moreover, as the code after LCS is found un-contained in the FSMD of golden program, this means that there is some code extra in the FSMD of student's program which follows the LCS and spans at least upto the cut-point next to LCS. This extra code block needs to be removed from the student's program.

Examples of cases 1-3

In the figures 2.50 to 2.53, a, a', b, b', c, c', d and d' represent the sets of data-transformations along the respective transitions/edges between the states.

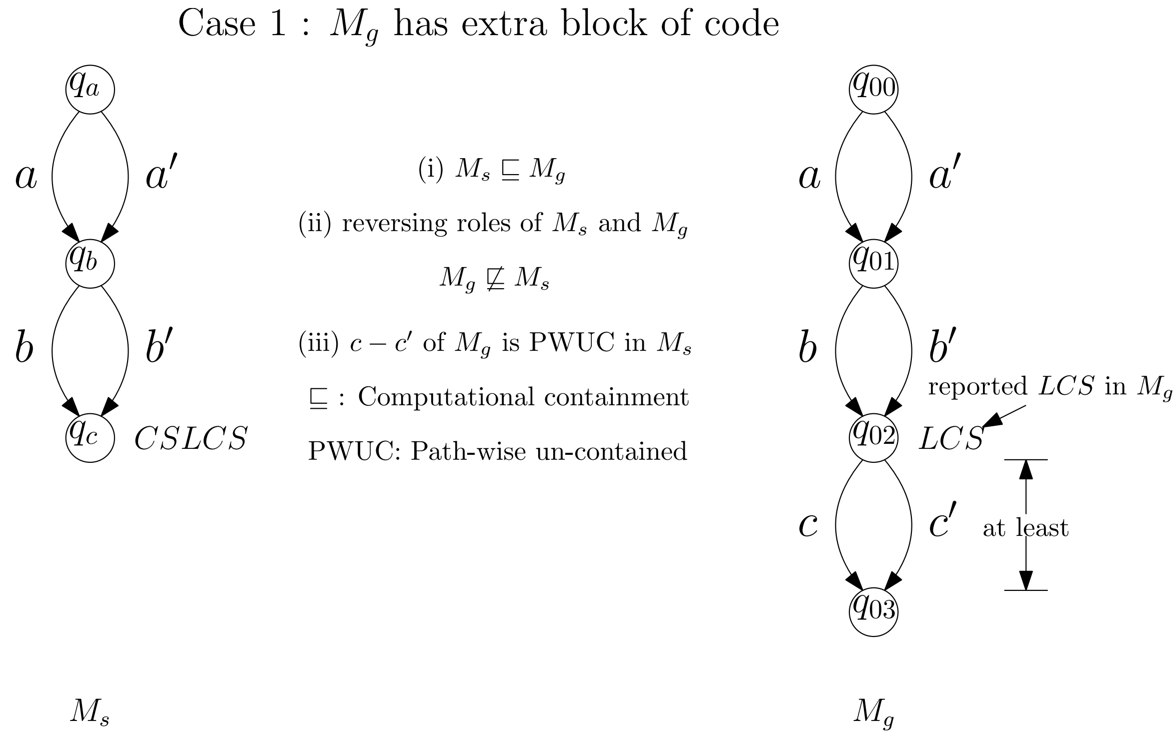
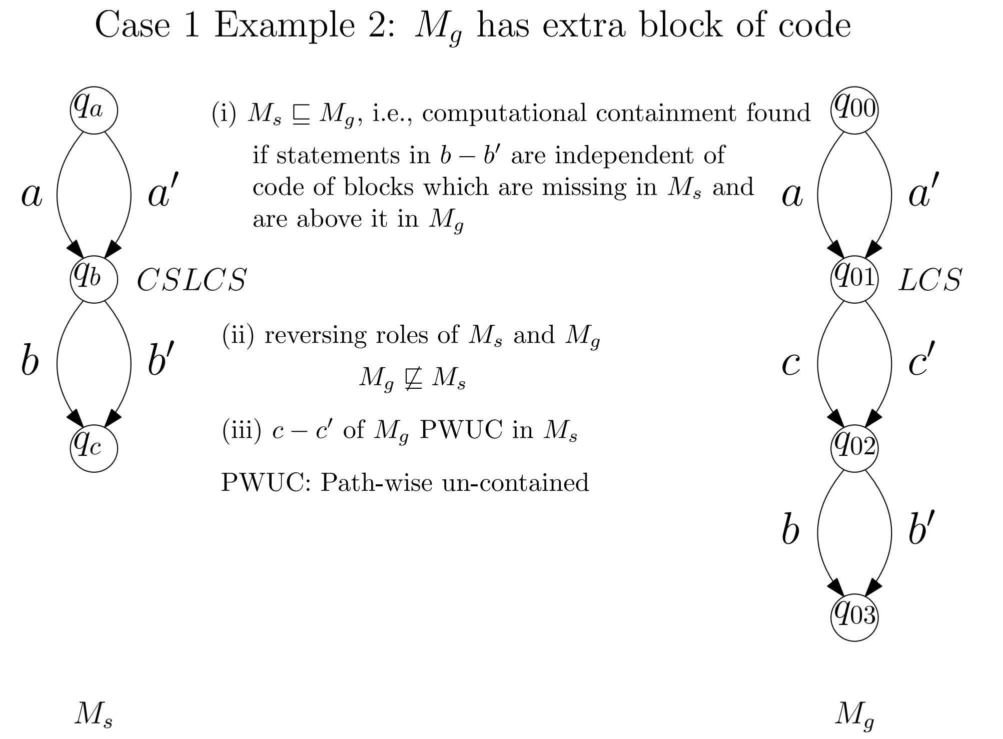

Case 1 : M_g has extra block of code
q_a
a
a'
q_b
b
b'
q_c
$CSLCS$
(i) $M_s \sqsubseteq M_g$
(ii) reversing roles of M_s and M_g
$M_g \not\sqsubseteq M_s$
(iii) $c - c'$ of M_g is PWUC in M_s
$\sqsubseteq$: Computational containment
PWUC: Path-wise un-contained
q_{00}
a
a'
q_{01}
b
b'
reported LCS in M_g
q_{02}
LCS
c
c'
at least
q_{03}
M_s
M_g

Figure 2.50: M_g and M_s, M_s has a missing block of code



Case 1 Example 2: M_g has extra block of code

(i) $M_s \sqsubseteq M_g$, i.e., computational containment found if statements in $b - b'$ are independent of code of blocks which are missing in M_s and are above it in M_g

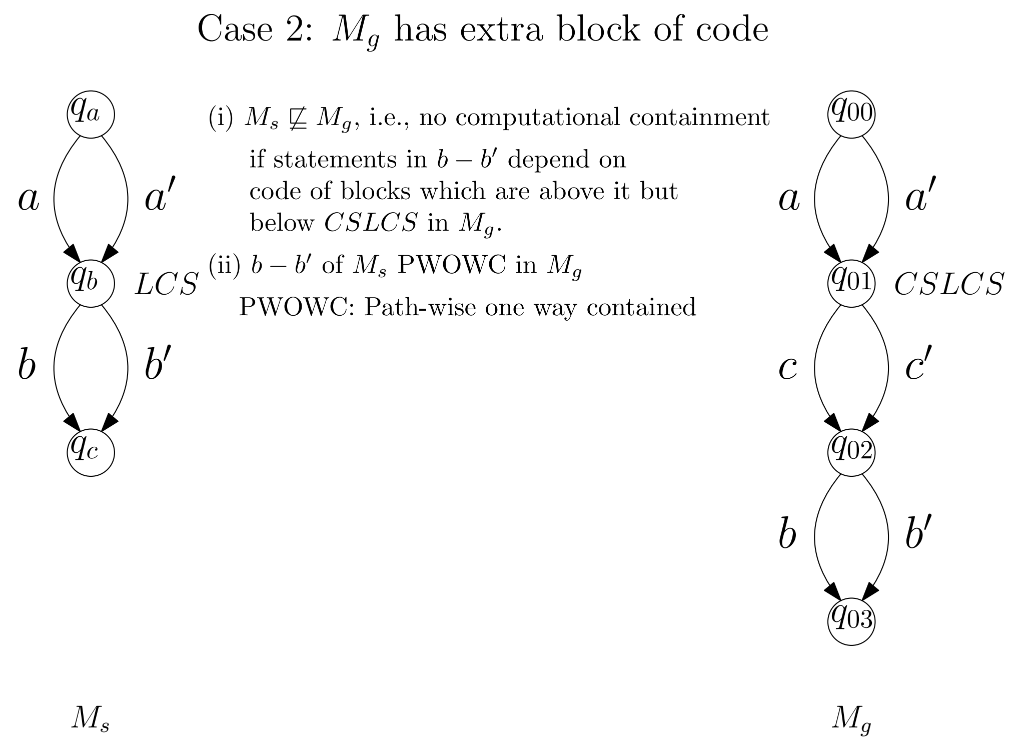
(ii) reversing roles of M_s and M_g

$M_g \not\sqsubseteq M_s$

(iii) $c - c'$ of M_g PWUC in M_s

PWUC: Path-wise un-contained

Figure 2.51: M_g and M_s, M_s has a missing block of code



Case 2: M_g has extra block of code

(i) $M_s \not\sqsubseteq M_g$, i.e., no computational containment if statements in $b - b'$ depend on code of blocks which are above it but below $CSLCS$ in M_g.

(ii) $b - b'$ of M_s PWOWC in M_g

PWOWC: Path-wise one way contained

Figure 2.52: M_g and M_s, M_s has a missing block of code

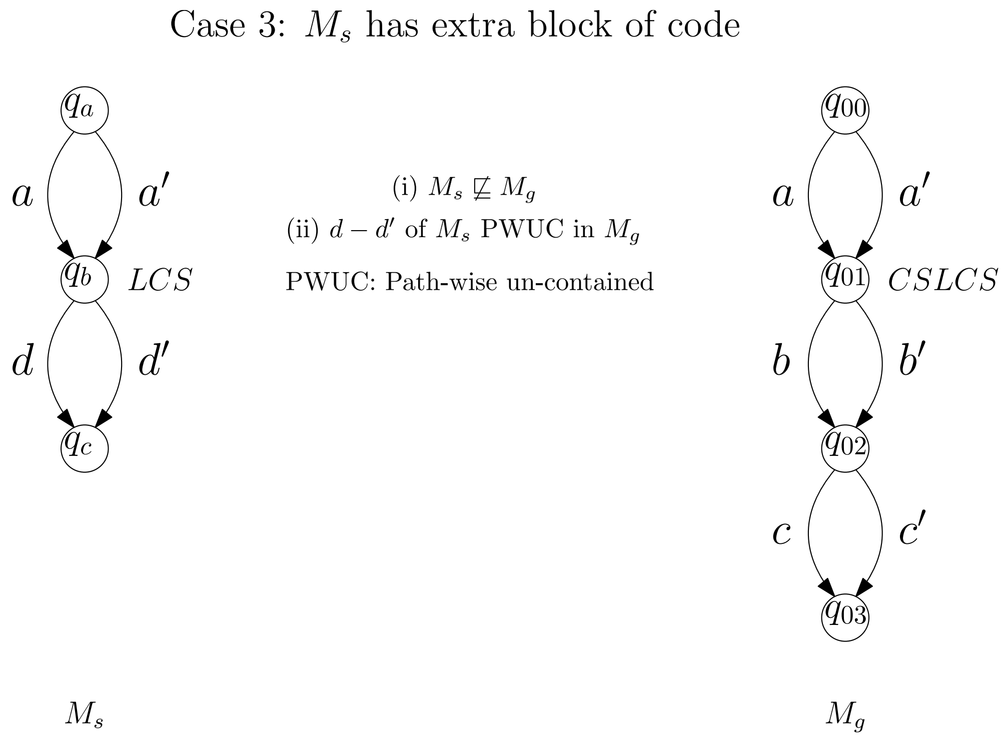

Case 3: M_s has extra block of code
q_a
a
a'
q_b
LCS
d
d'
q_c
(i) $M_s \not\subseteq M_g$
(ii) $d - d'$ of M_s PWUC in M_g
PWUC: Path-wise un-contained
$q00$
a
a'
$q01$
$CSLCS$
b
b'
$q02$
c
c'
$q03$
M_s
M_g

Figure 2.53: M_g and M_s, M_s has extra block of code

Example of combined cases 1-3

In the figures 2.54 to 2.57, a, a', b, b', c, c', d and d' represent the sets of data-transformations along the respective transitions/edges between the states. These figures show how removing the extra code block is done. The missing code blocks will then be located as discussed in case 1 or 2, as the case may be. The missing blocks are then introduced using the procedure given subsequently in this section.

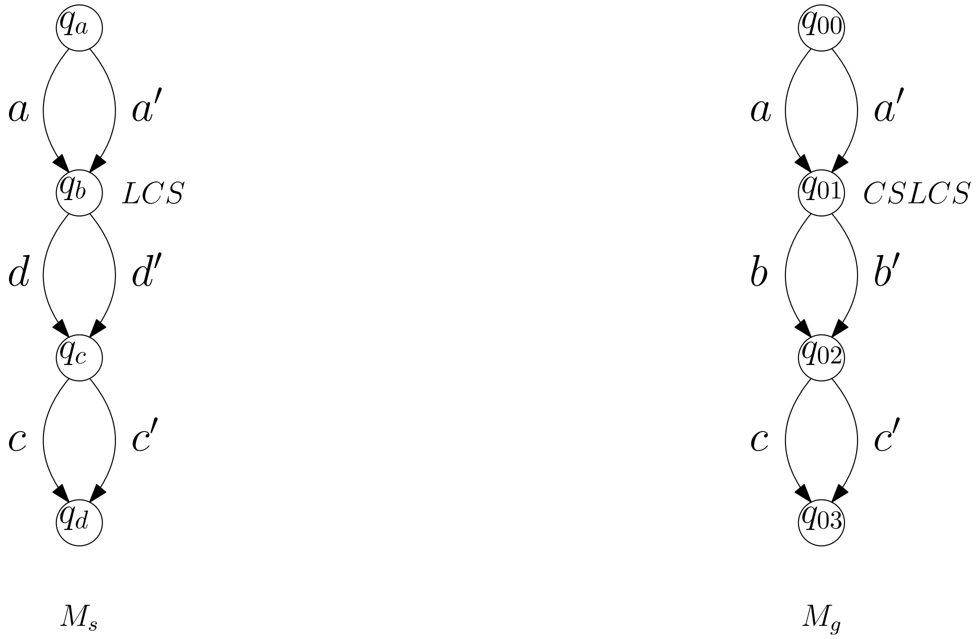


Figure 2.54: M_g and M_s, M_s has a missing block and an extra block of code

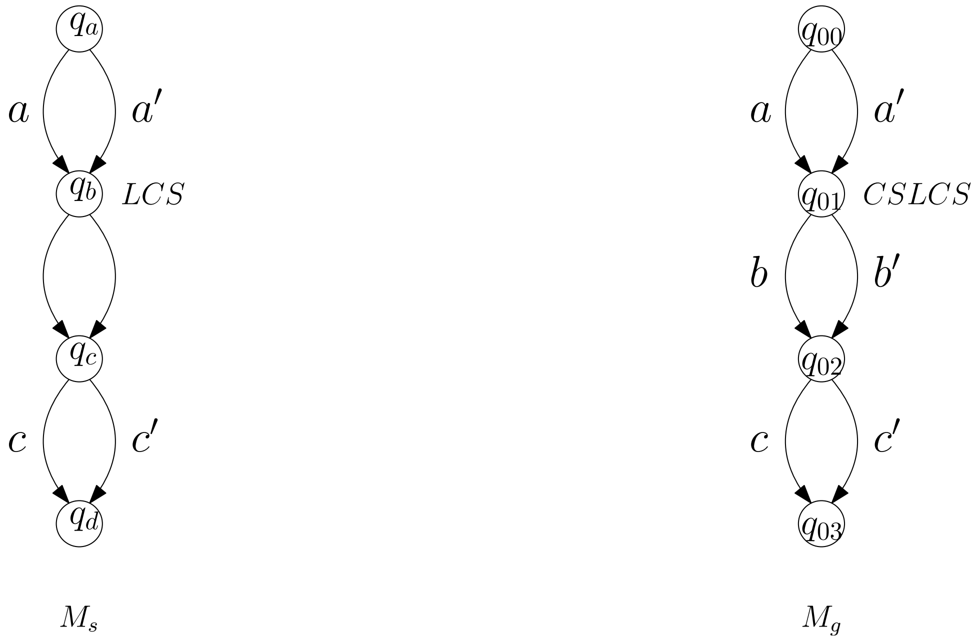


Figure 2.55: M_g and M_s, removing code from extra block in M_s

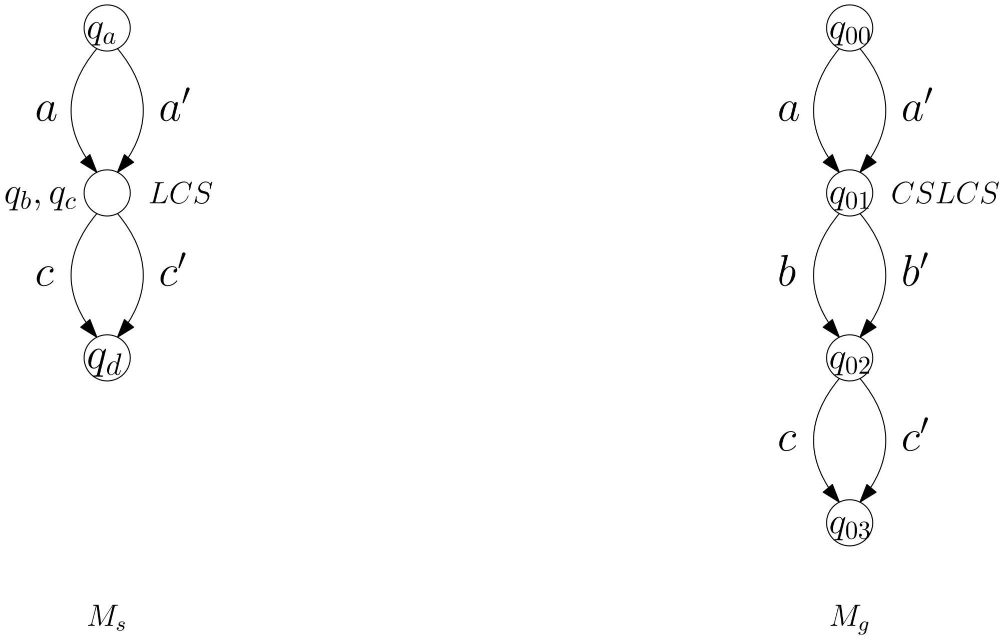


Figure 2.56: M_g and M_s, merging the states at the two ends of extra block in M_s

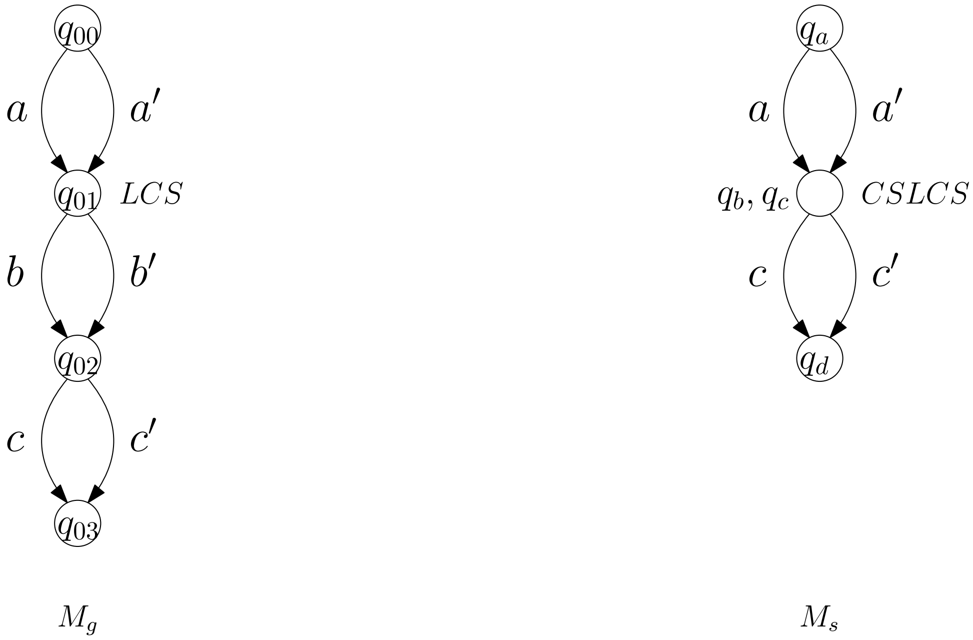


Figure 2.57: M_g and M_s reversed, equivalence check finds $M_g \not\sqsubseteq M_s$

We present below the definition of correction vector, as the term has been used in several places in the following discussion.

Definition 13 (Correction vector)**.** *The correction vector for each state S_i of an FSMD* *is an n-tuple* $t_i = \langle t_{i,1}, t_{i,2}, \ldots, t_{i,n} \rangle$*, where n is the number of outgoing transition from the state, each* $t_{i,k}, 1 \leq k \leq n$ *is an outgoing transition from the state* S_i*.*

Definition 14 (Incoming vector)**.** *The incoming vector for each state* S_i *of an FSMD is an n-tuple* $t_i = \langle t_{i,1}, t_{i,2}, \ldots, t_{i,n} \rangle$*, where n is the number of incoming transition to the state, each* $t_{i,k}, 1 \leq k \leq n$ *is an incoming transition to the state* S_i*.*

Steps for introducing extra code

(i) The correction vector C_v, for each state on FSMDs M_g and M_s, are stored in a 2-dimensional correction vector array, one each for M_g and M_s.

(ii) The incoming vector I_v, for each state on FSMDs M_g and M_s, are stored in incoming vector linked lists, one each for M_g and M_s.

(iii) Find LCS (*last correct state*) and CSLCS (the corresponding state of the *last correct state* in the correct FSMD).

(iv) If the correction vector of the CSLCS, C_v [CSLCS], is the same as the outgoing transitions from the LCS, then exit. Let the node v in the FSMD be the CSLCS, then it's correction vector is C_v. If C_v is the same as the outgoing transitions from the LCS, then exit.

(v) Introduce a state above LCS and make the outgoing transitions of the newly introduced state, the same as the ougoing transitions of CSLCS. The transitions previously incident on LCS will now be made incident on newly introduced state. The outgoing transitions from the newly introduced state will be made incident on the LCS. Now the new corresponding state of LCS will be the state, which is immediately below the CSLCS.

(vi) Find out the set difference of incoming edges on the CSLCS and those on the LCS. Push all the edges in the set difference on $stack1$.

(vii) Find out the set difference of incoming edges on the LCS and those on the CSLCS. Push all the edges in the set difference on $stack2$.

(viii) If the content of both the stacks are not empty and the top of both the stacks are the same, then we pop both the stacks. The popped out edge is the edge which is to be correctly placed on the modified incorrect FSMD, as it is placed in the golden program's FSMD. Also output the code starting from the start state of popped out edge, to the state at the other end of the popped out edge, as the code which should be properly placed as per the golden program.

(ix) Add the incoming edges of the newly introduced state and the newly introduced state itself to the incoming vector table for M_s. Go to **step *(iii)***.

Note: When the missing condition is not nested inside another condition, then the same mechanism discussed above will work except for manipulations of $stack1$ and $stack2$, they will not play any role.

The FSMD of golden program having all the blocks is shown in the Figure 2.58. In the Figure 2.59, the FSMD of student's incorrect program is shown. The student's program is missing a block of code, which is to be inserted.

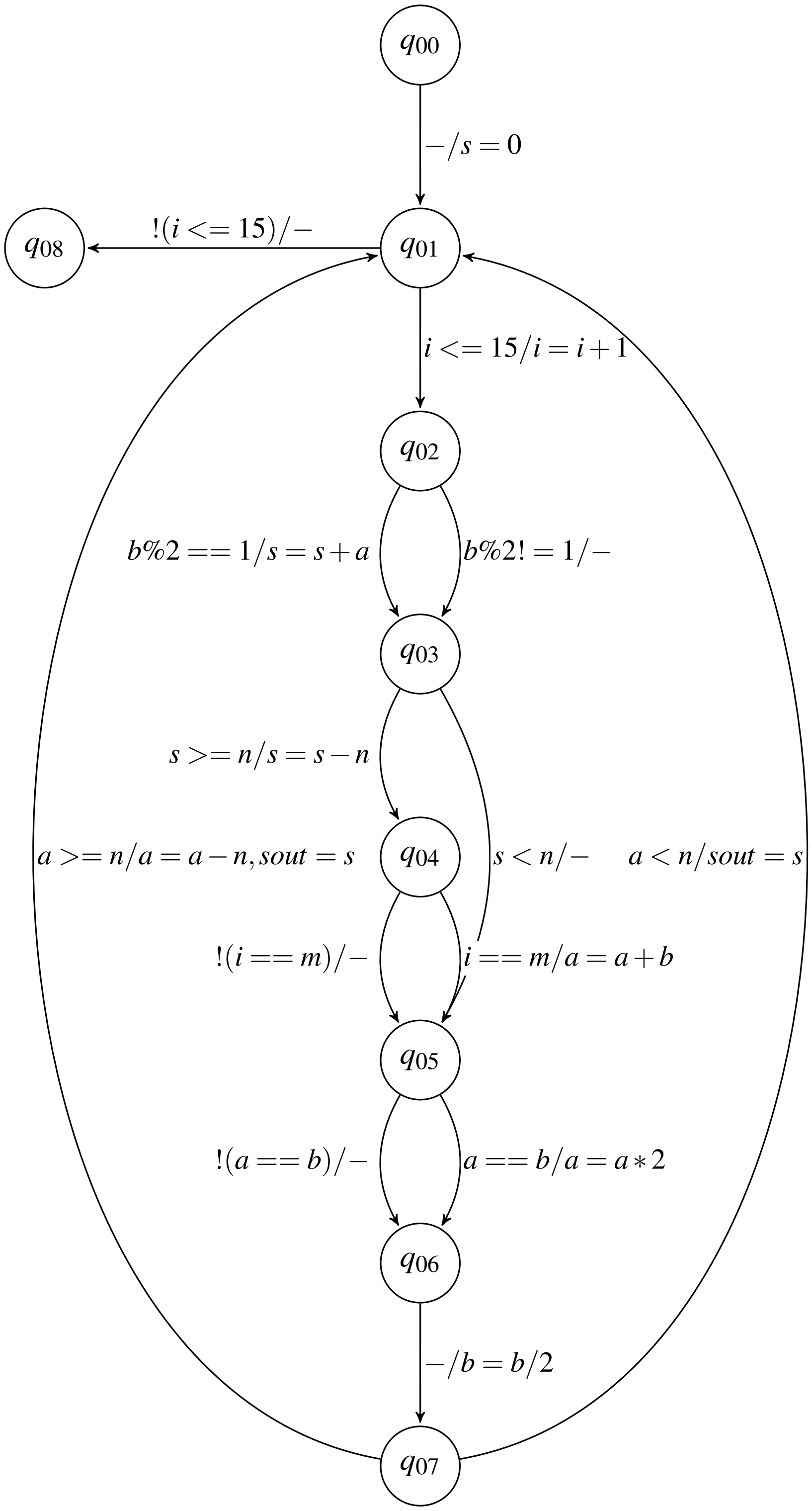


Figure 2.58: Figure showing M_g with several blocks of code.

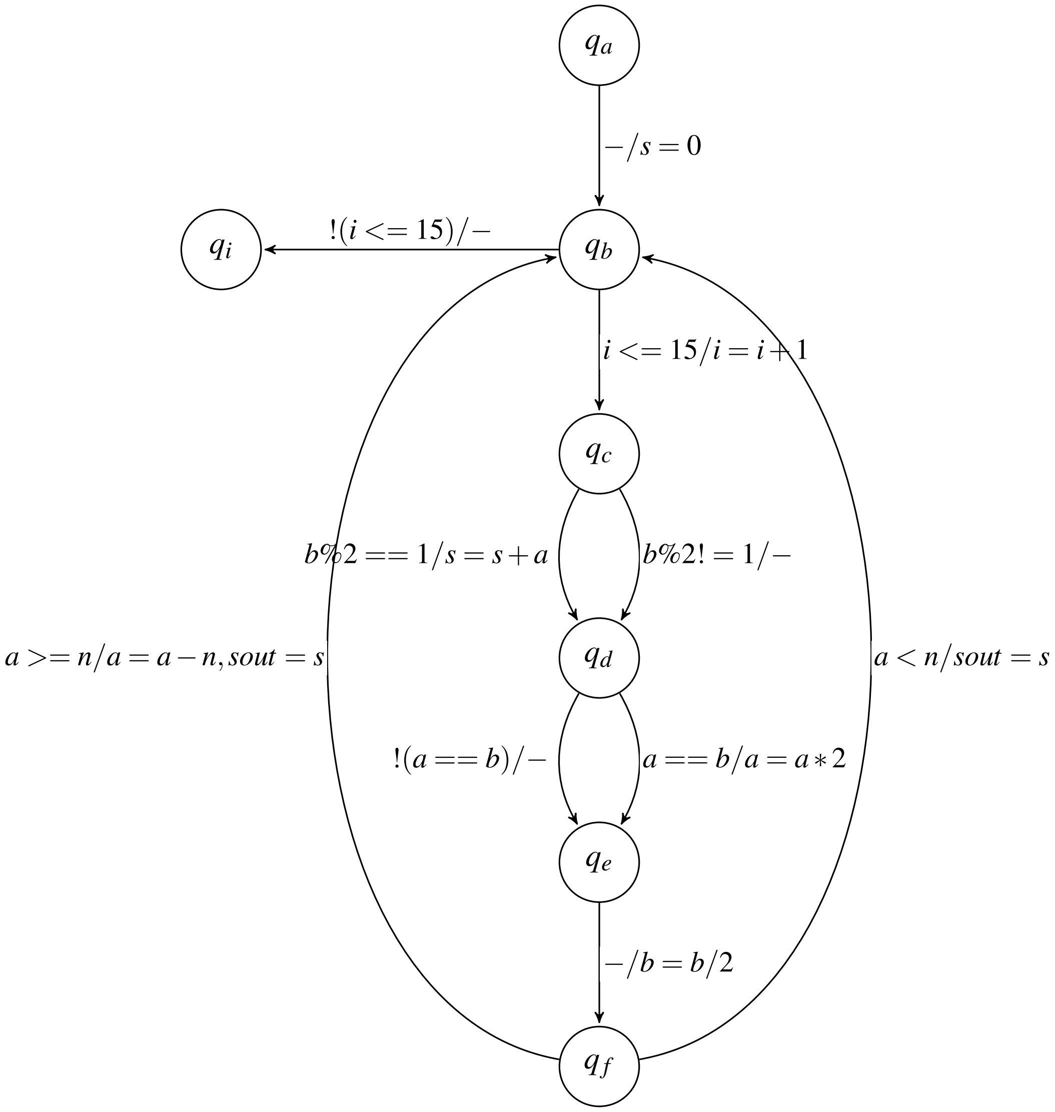


Figure 2.59: Figure showing M_s with missing block of code.

As per the equivalence checker's report, the LCS is found at q_d and CSLCS at q_{03} in the step 3. In step 4, comparison of the outgoing transitions of reveals that they are not the same. In step 5 a state q'_{03} is introduced above q_d, the LCS, with its outgoing transitions being the same as those of CSLCS, and they are made incident on LCS. This is shown in the Figure 2.60.

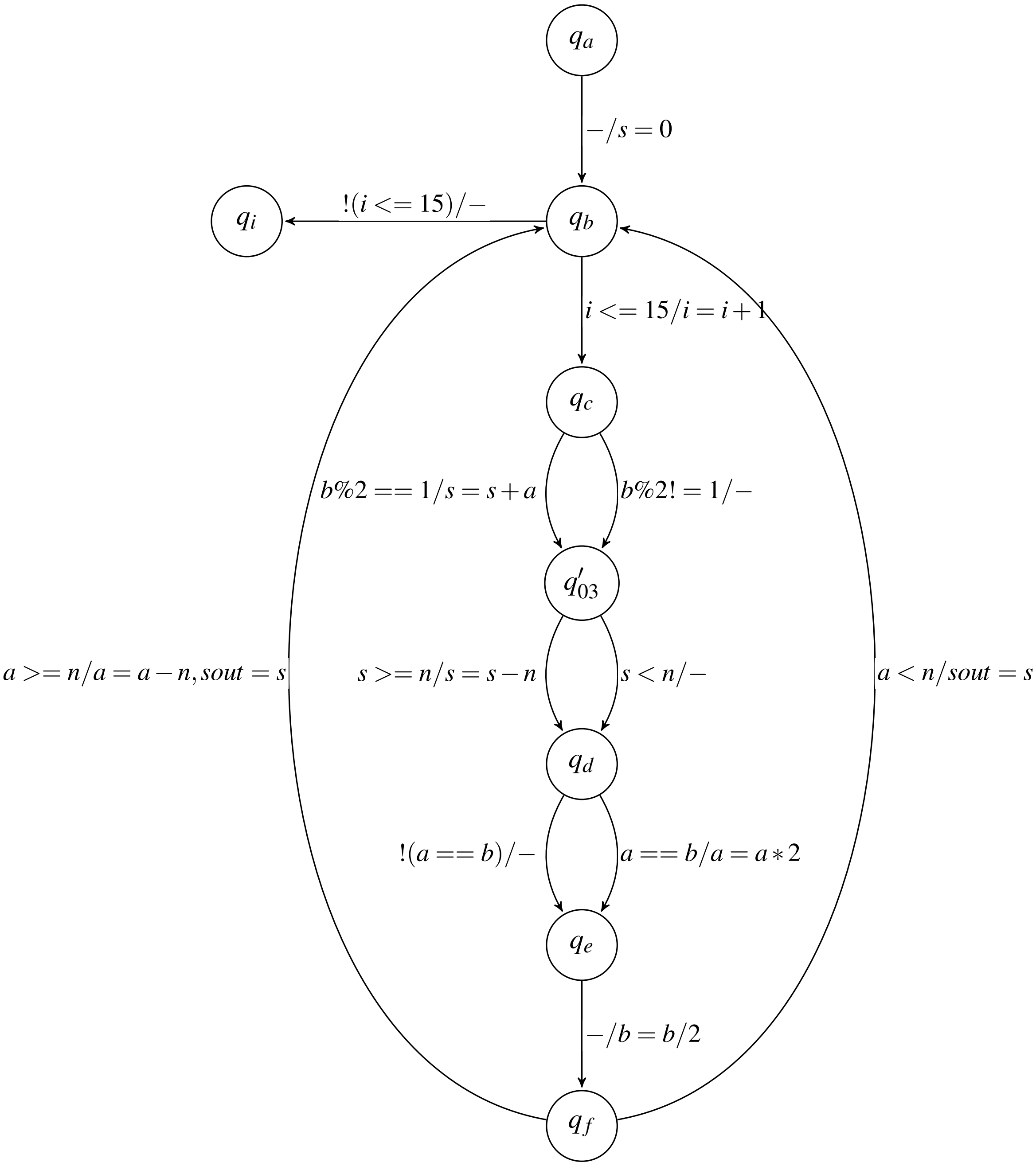


Figure 2.60: Figure showing M_s with application of correction vector.

In the second iteration, step 3 finds that now q_{04} has become the CSLCS, LCS is the same. Their outgoing transitions are not the same so a new state q'_{04} is introduced above q_d, the LCS giving the following FSMD in the Figure 2.61.

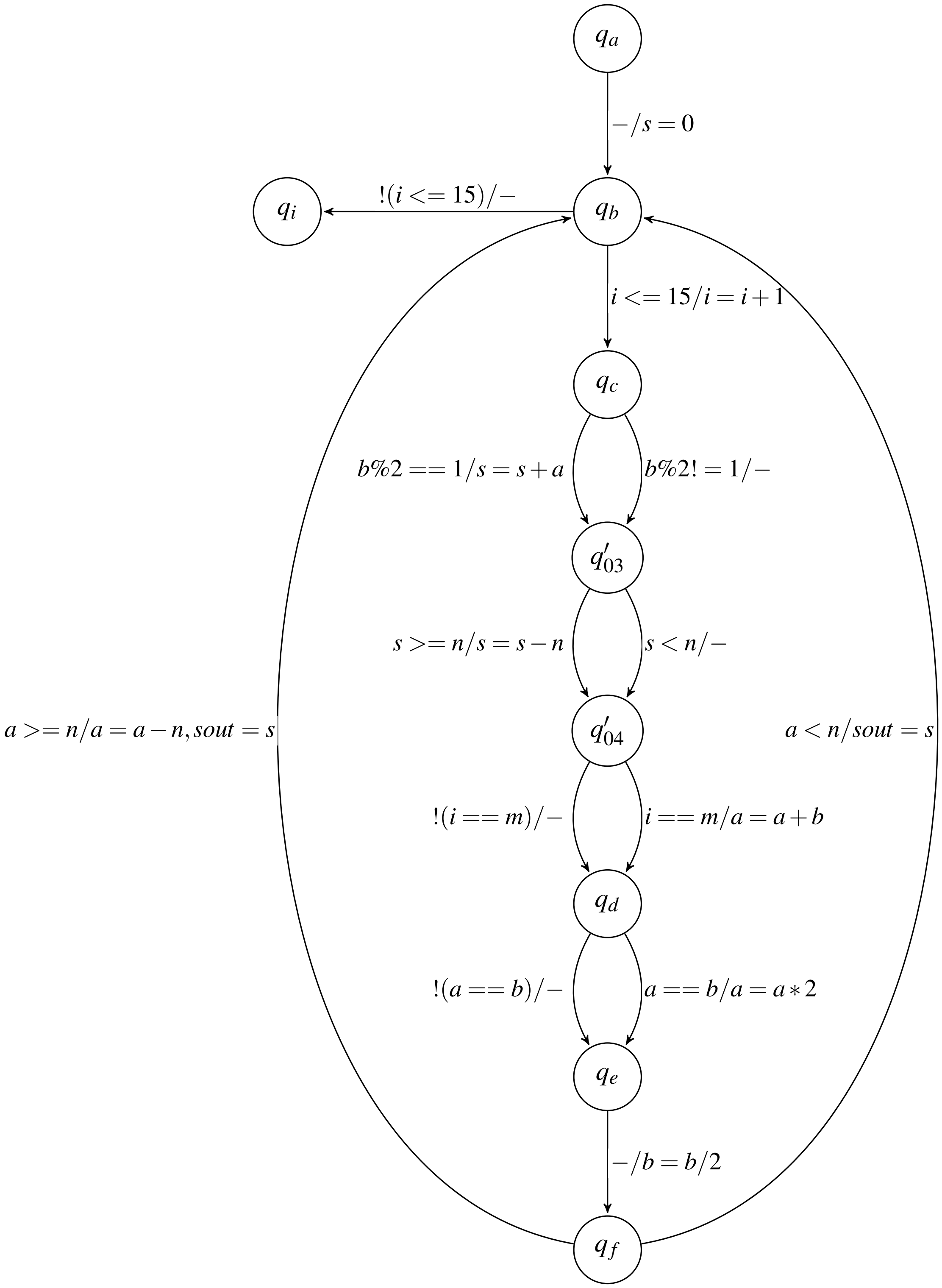


Figure 2.61: Figure showing M_s with application of correction vector for the second time.

Proper placement of the edges is visible in the next step after the execution of the steps of computing the set difference of incoming edges and because of equal entries at the top in stack1 and stack2, resulting in the proper positioning of the edges. This is shown in the Figure 2.62

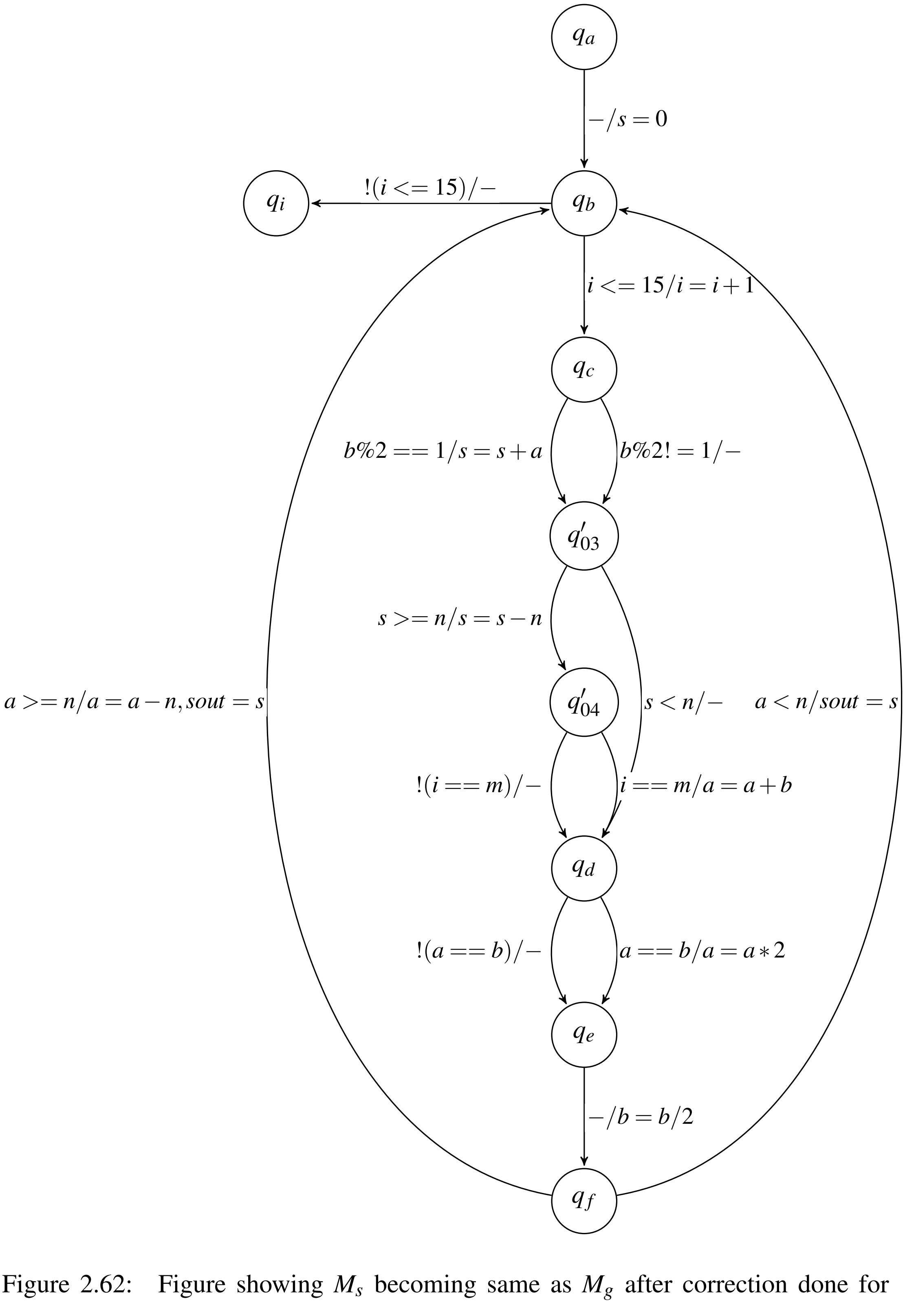


Figure 2.62: Figure showing M_s becoming same as M_g after correction done for skipping parenthesis.

The above mechanism to introduce code uses a 2-stack method, which works if the structures, shown in the examples for various cases, is present. For other structures, we have another mechanism to introduce code, which is given in sections 2.3.6 and 2.3.9.

Unified algorithmic steps for automated assessment

1. Execute the equivalence checker with the following parameters:

   (a) M_s, FSMD file of student's program and

   (b) M_g, FSMD file of teacher's program

2. If equivalence checker reports $M_s \not\sqsubseteq M_g$, then execute the containment checker with LCS as a parameter.

3. **Switch Case:**

   Case (Containment checker has reported unordered path-wise both way contained):

   check for dependency violation

   Case (Containment checker has reported path-wise one way contained):

   call the algorithm for missing code

   Case (Containment checker has reported path-wise un-contained):

   call the algorithm for removing code

   Case (Containment checker has reported unordered path-wise both way contained and path-wise one way contained for faulty branching):

   call the algorithm for parenthesis skipping

4. If equivalence checker reports $M_s \sqsubseteq M_g$, then interchange the FSMDS M_s and M_g and execute the equivalence checker again. If now the equivalence checker

reports $M_g \not\sqsubseteq M_s$, then execute the containment checker with LCS as a parameter. If the containment checker has reported path-wise un-contained, this means there is missing code, call algorithm for missing code.

In these steps, M_s is transformed to M_s' such that as much as possible of M_s is retained.

Algorithm 6: Unified_algorithm_for_automated_assessment

```
    Input: $M_s$, FSMD of student's program and $M_g$, FSMD of teacher's program;
    Output: Transform $M_s$ to $M_s'$ such that $M_s' \equiv M_g$
L1  Begin;
    /*Step 1:  Execute the equivalence checker with the following parameters:  (i)$M_s$ and (ii)$M_g$*/
L2  repeat
L3      if equivalence checker reports $M_s \not\sqsubseteq M_g$ then
L4          execute the containment checker with LCS as a parameter;
L5          switch Report of containment checker do
L6              case Containment checker has reported unordered path-wise both way contained do
L7                  check for dependency violation and incorporate corrections to remove dependency violations;
L8              case Containment checker has reported path-wise one way contained do
L9                  call the algorithm for inserting the missing code in $M_s$;
L10             case Containment checker has reported path-wise un-contained do
L11                 call the algorithm for removing code from $M_s$ i.e.,;
L12                 Remove all the data-transformations from $p_s$ and $p_s'$ (i.e., the extra block $p_s - p_s'$ in $M_s$);
L13                 Merge the two states at the two ends of both $p_s$ and $p_s'$;
L14             case Containment checker has reported unordered path-wise both way contained and path-wise one
                  way contained for faulty branching do
L15                 call the algorithm for parenthesis skipping;
L16     else
L17         interchange the FSMDs $M_s$ and $M_g$ and execute the equivalence checker again;
L18         if equivalence checker reports $M_g \not\sqsubseteq M_s$ then
L19             execute the containment checker with LCS as a parameter;
L20             if the containment checker has reported path-wise un-contained then
L21                 this means there is missing code, call the algorithm for inserting the missing code in $M_s$;
L22 until equivalence checker reports $M_s \sqsubseteq M_g$&&$M_g \sqsubseteq M_s$;
L23 End
```

Prior to line 13, the code in the if part corresponds to the condition $M_s \not\sqsubseteq M_g$. Line

13 comes under the else part ($M_s \sqsubseteq M_g$), where there are two possibilities,

1. either $M_g \sqsubseteq M_s$ in which case $M_s \equiv M_g$,
2. or $M_g \not\sqsubseteq M_s$.

The line 13 of the algorithm is to explore the latter possibility, which means that as $M_s \sqsubseteq M_g$, but $M_g \not\sqsubseteq M_s$ and thus $M_s \not\equiv M_g$, then it must be the case that transitions in M_g are a superset of those in M_s. In other words the golden program has some extra code as compared to the student's program, which means student's code is missing some lines of code. This is the way to ascertain missing code in the student's program.

In other words, in line 2, $M_s \not\sqsubseteq M_g$ means that all computations in M_s are not contained in M_g, but still there may be the case that all coputations in M_g are contained in M_s. This possibility is explored in line 13.

2.3.4 Error of missing code in the nested cases

In this part of work, we have handled the case of missing code inside student code. There can be many cases of errors of missing code in a program. If there is a nested loop structure required in a program, a student is likely to miss some nested loop. Other possibility of error is when there are many conditions to be checked in a program, it is highly possible that student can miss some cases. Nested condition also gives rise to possibility of missing code. Student's errors are likely to miss some nested condition and the associated code. In the subsection 2.3.5 given below, we have discussed the implementation approach for missing loop. We consider the case in which there are two nested loops but outer loop is missing. In subsection 2.3.8, we have discussed the case, when a nested condition is missing. The steps for implementation of correction mechanism are given in subsection 2.3.9.

2.3.5 Missing the code of nested loop

In this section we discuss the approach for handling the case of missing code of nested loop. This case arises when there is nested loop in correct program but student has missed out some nested loop in the program. This will make the FSMD of student program different from teacher's program. For handling this type of error we have to introduce the loop in the program. Equivalence checker gives the path inside student program for which it is failing. Starting of this path will be our LCS. Now the containment checker reports the path in teacher's FSMD, which contains the reported path of equivalence checker. The output of containment checker gives us the intuition that path of student's FSMD for which equivalence checker is failing, is present in teacher's FSMD, but there is some extra code also along this path. So in order to introduce extra code we need to know the correct state inside student's FSMD, where we can apply the correction vector. This state will be some state which generally comes on some path after the LCS. Such a state is the state before which the equivalence actually exists in the two FSMDs. This is so because as the LCS is the last cut-point state up to which equivalence is established, there may still be some transitions beyond LCS, which may be equivalently present on both the FSMDs examined for equivalence. Thus, after LCS, there may be a non cut-point state, up to which the two FSMDs will have equivalence. Such a non cut-point state will play an important role in introducing the missing code of the nested loop, which we will see in the steps for correction mechanism given later in this description. It may further be noted that the state corresponding to such a non cut-point state found beyond LCS, may be a cut-point state in the other FSMD. We present such a case in the discussion of correction mechanism given below.

Below we have drawn FSMDs of programs which calculate the sum of digit and again sum of digits of sum until we get a single digit number. First figure 2.63 shows the FSMD corresponding to correct program (written below the figure 2.63). Second figure 2.64 is the FSMD corresponding to student program. Student has missed one outer while loop. He has calculated sum of the digit only once, making his program incorrect. The corresponding to program is written below the figure 2.64.

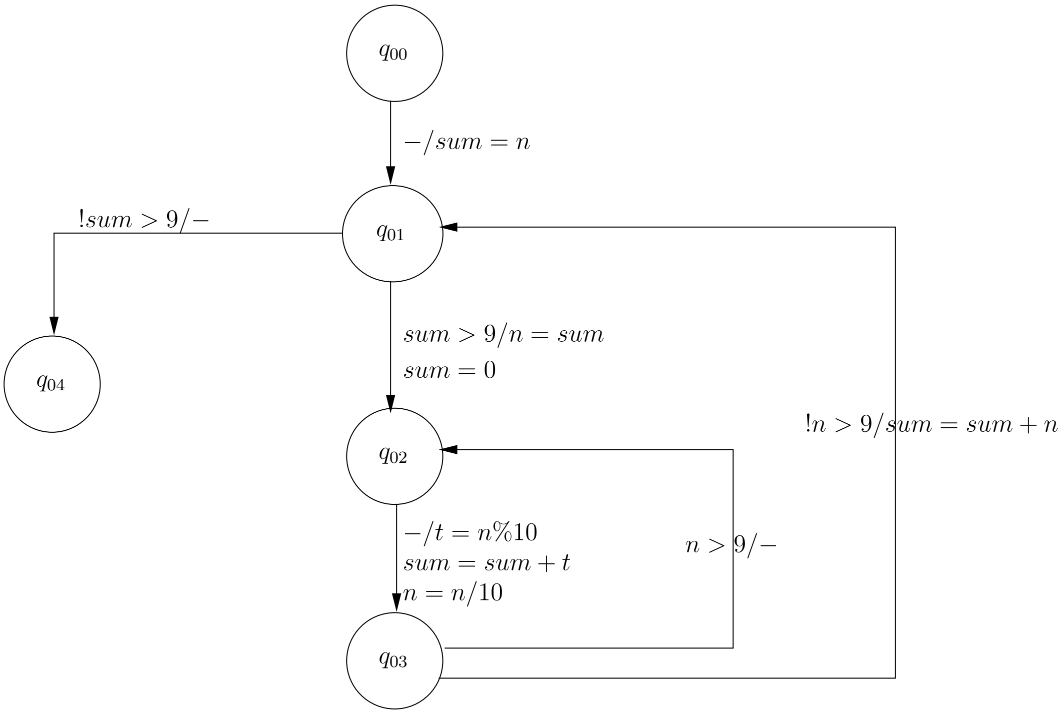


Figure 2.63: An M_g for digitsum program

digitsum_correct.c

```
#include<stdio.h>
int main() {
       int n, i, sum, t;
       printf("enter the number");
       scanf("%d", &n);
       sum = n;
       while (sum > 9) {
           n = sum;
           sum = 0;
           do {
                t  =n % 10;
                sum = sum + t;
                n = n / 10;
               } while (n > 9);
```

```
            sum = sum + n;
       }
        printf("%d", sum);
}
```

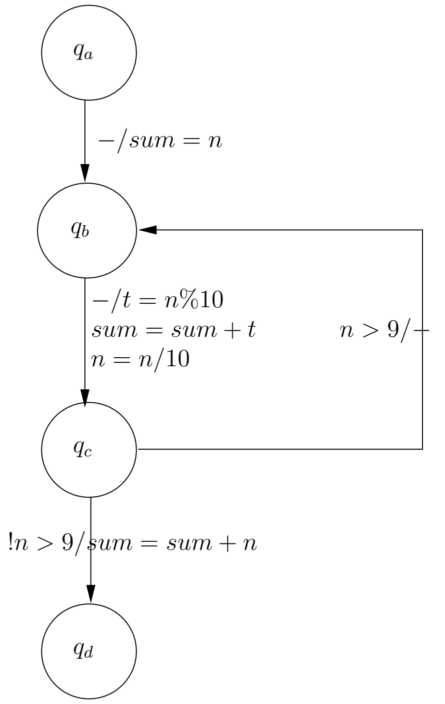


Figure 2.64: An M_s for digitsum program.

digitsum_incorrect.c

```
#include<stdio.h>
int main() {
     int n, i, sum;
     printf("enter the number");
     scanf("%d", &n);
     sum = n;
     do {
          sum = sum + (n % 10);
          n = n / 10;
     } while (n > 9);
     sum = sum + n;
     printf("%d", sum);
}
```

The correction mechanism

Equivalence checker fails to find equivalence and give us q_a as the LCS. We, therefore, select the path from q_a to next cut point i.e., $q_a \rightarrow q_b \rightarrow q_c$. We find the containment of selected path, $q_a \rightarrow q_b \rightarrow q_c$, in the correct FSMD with the help of containment checker. Containment checker outputs **path-wise one way contained** and the containing path is $q_{00} \rightarrow q_{01} \rightarrow q_{02} \rightarrow q_{03}$. This gives us the intuition that although the contents of the path $q_a \rightarrow q_b \rightarrow q_c$ are present in correct FSMD, there is some extra code also in the correct program, which is missing in the incorrect program on the path $q_a \rightarrow q_b \rightarrow q_c$. We have to introduce the missing code. We find the exact state in incorrect FSMD, where we have to put that extra code. In order to find this, we first check whether the equivalent path of the path from CSLCS to the next state, that is $q_{00} \rightarrow q_{01}$, is present in the incorrect FSMD. It is found to be so. As the missing state has not yet been found, we go further by checking presence of equivalent path of the path from q_{01} to the next state, i.e., $q_{01} \rightarrow q_{02}$, inside the incorrect FSMD, which is not found there. Hence, we conclude that the state corresponding to q_{01} is the missing state in the incorrect FSMD, as up to q_{01} we could find that the corresponding paths existed in the incorrect FSMD. We then check whether q_{01} is a state corresponding to loop and we find that it is actually so, by using a DFS based mechanism. Simply introducing the loop won't work, as we have to adjust the correct nesting of the loop. For that, we keep on pushing the code below newly introduced loop, to the loop belonging to the newly introduced loop state, until we get the loop-back transition coming to newly introduced state, equivalent to loop-back transition coming to corresponding state of newly introduced state.

2.3.6 Steps for correction mechanism

Let us use the following notations

(1) OutTrans: outward transition from a loop entry state going out of the loop,

(2) InTrans: inward transition from a loop entry state going inside the loop.

In a transition $t_{i,e}$ going from the state s_i to the state s_e, s_i is called the *start state* and s_e is called the *end state* of the transition.

We use a structure having the following members.

(1) OutTransEndState: to hold the end state of OutTrans,

(2) InTransEndState: to hold the end state of InTrans,

(3) StateCondition: to hold the loop condition at a loop entry state.

(4) StateMg: to hold the loop entry state of M_g.

(5) StateMs: to hold the loop entry state of M_s.

The stack StateStack is a stack containing each of its elements as a structure with the members as stated above. StateStack is used in the steps given below.

The terms are clarified in the figure 2.65.

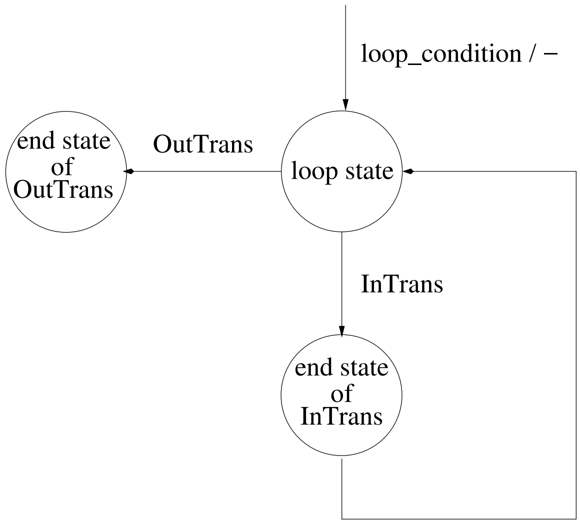


Figure 2.65: Outgoing transitions from loop state

(1) Run the equivalence checker on the FSMD M_s of student's program and FSMD M_g of teacher's program. In case the programs are not equivalent, find the last correct state (LCS) in M_s and the corresponding state CSLCS in M_g.

(2) Run the containment checker, let p_s be the path of FSMD M_s (starting from LCS) for which the containment checker yields "path-wise one way contained"; this means there is a path p_g (may be an extended path), starting from CSLCS in M_g, which contains some extra transitions in addition to all the transitions of p_s.

(3) Start from the LCS in M_s and CSLCS in M_g. Find the first transition t_g in p_g which is not matching with p_s. The first state in p_s from which the t_g did not match will henceforth be called an *unmatched state* (US). The state in p_g, which corresponds to the state US, is called *the corresponding state of the unmatched state* (CSUS).

The following cases are possible.

(i) CSUS is not a cut-point, let s_g be the only successor state of CSUS.

(ii) CSUS is a cut-point, but not a loop entry state, let s_g be the successor state on either of the outgoing transitions from CSUS.

(iii) CSUS is a cut-point and also a loop entry state, let s_g be the successor state of CSUS along the transition leading inside the loop.

Initially, let s_s be same as US. We have the following action for the above cases.

(4) **Repeat:**

```
Switch Case:
  Case (CSUS is not a cut-point):
      let $s_g$ be the only successor state of CSUS
      call the function for case I
  Case (CSUS is a cut-point, but not a loop entry state):
      let $s_g$ be the successor state on either of the outgoing transitions from
         CSUS
      call the function for case II
  Case (CSUS is a cut-point and a loop entry state):
      let $s_g$ be the successor state of CSUS along the transition leading inside
         the loop
      call the function for case III
Until (there is no state after $s_g$)
```

(1) **Function for case I:** The CSUS is neither a cut-point nor a loop entry state.

(I) Let *ChainMg* be the sequence of all the transitions from s_g to the next cut-point;

(II) Let *ChainMs* be the sequence of all the transition from s_s to the next cut-point;

(III) Let *chain* = Equate_chain(*ChainMg*, *ChainMs*) // *ChainMs* will become same as *ChainMg*;

Push *chain* to at s_s. The end state of *chain* will be s_s and end state of *ChainMg* will be s_g.

(2) **Function for case II:** If CSUS is a cut-point, but not a loop entry state.

Execute the copy mechanism for non-loop cut-points, given at the end of this section, with s_g and s_s as CSLCS and LCS respectively. The last states acted upon by executing the copy mechanism are the states of M_g and M_s where the correction vector (outgoing transitions) were found equal. These states will be now s_g and s_s in M_g and M_s respectively.

(3) **Function for case III:** The CSUS is a cut-point and also a loop entry state.

(I) Introduce a self-loop entry state s_n before US, with loop condition of CSUS.

- Loop condition at s_n will be same as the loop condition at CSUS.
- The condition for outward transition from s_n will be the negation of loop condition at CSUS. The outward transition from s_n will be incident on US.
- Incoming transition on s_n will be the one which was earlier incoming transition at US.

(II) Push the following information into StateStack.

- OutTransEndState = US
- InTransEndState = US
- StateCondition = Loop condition of CSUS.

(III) **Repeat**

Switch Case:

Case (If s_g is a loop entry state and s_s is also a loop entry state and loop condition at s_g and loop condition at s_s matches):

(A) Push the following information to StateStack

- OutTransEndState = OutTrans of s_s

- InTransEndState = InTrans of s_s
- StateCondition = Condition of s_s

(B) Introduce a self loop entry state, s_t, inside the loop of s_n. The loop condition of s_t will be the loop condition of s_s.

- The incoming transition on s_t will be the inward transition of s_n.
- The condition of outward transition from s_t will be the negation of loop condition of s_g. The outward transition from s_t will be incident on s_n.
- Set $s_n = s_t$, s_s = InTransEndState of top(StateStack).

Case (If s_g is a loop entry state and s_s is also a loop entry state and loop condition at s_g and loop condition at s_s does not match):

(A) Introduce a self loop entry state, s_n, before s_s

- Loop condition at s_n will be same as the loop condition at s_g.
- The condition for outward transition from s_n will be the negation of loop condition at s_g. The outward transition from s_n will be incident on s_s.
- Incoming transition on s_n will be the one which was earlier incoming transition at s_s.

(B) Push the following information into StateStack.

- OutTransEndState = s_s
- InTransEndState = s_s
- StateCondition = Loop condition of s_g.

Case (If s_g is cut-point but not a loop entry state):

Execute the copy mechanism for non-loop cut-points, given at the end of this section, with s_g and s_s as CSLCS and LCS respectively. The last states acted upon by executing the copy mechanism are the states of M_g and M_s where the correction vector (outgoing transitions) were found equal. These states will be now s_g and s_s in M_g and M_s respectively.

Case (If s_g is neither a cut-point nor a loop entry state):

(A) Let us use the following notations

- $s_{condStack,g}$ be the state of M_g at *StateMg* of top(StateStack)

- $s_{condStack,s}$ be the state of M_s at $StateMs$ of top(StateStack)
- $ChainMg$: The sequence of all the transition from $s_{condStack,g}$ to the next cut-point in the loop at $s_{condStack,g}$.
- $ChainMs$: The sequence of all the transition from $s_{condStack,s}$ to the next cut-point in the loop at $s_{condStack,s}$.

(B) *chain* = Equate_chain(*ChainMg*, *ChainMs*) // *ChainMs* will become same as *ChainMg*

(C) Push *chain* to the loop at s_n. The end state of *chain* will be s_s and end state of *ChainMg* will now be s_g.

Case (If feedback of s_n becomes equal to the feedback of s_g):

- s_g = OutTrans of s_n.
- s_s = OutTransEndState of top(StateStack).

Pop StateStack.

Until StateStack is not empty

The chain copy mechanism function Equate_Chain

The steps of the mechanism are as follows.

```
Given a state $s_g$ of $M_g$ and a state $s_s$ of $M_s$
If $s_g$ and $s_s$ are not cut-points
     Make $ChainMg$
     Make $ChainMs$
     ForAll states $s_t$ of $ChainMg$ starting from $s_g$
          ForAll states $s_{tc}$ of $ChainMs$ starting from $s_s$
              If $s_t == s_{tc}$
                  continue
                 Else
                  Search $s_t$ in $ChainMs$
                  If $s_t$ is found in $ChainMs$
                     Place it as it corresponds to $s_g$
```

```
Else $s_g$ not found
    Introduce a state similar to $s_g$ at $s_s$
```

Copy mechanism for non-loop cut-points

1. Find LCS and CSLCS. In case CSLCS is cut-point, not a loop start state, corresponding to if - else condition do the following.

2. If the correction vector of the CSLCS, C_v [CSLCS], is the same as the outgoing transitions from the LCS, then exit.

3. Introduce a state above LCS and make the outgoing transitions of the newly introduced state, the same as the outgoing transitions of CSLCS. The two transitions will correspond to a condition and its negation. The transitions previously incident on LCS will now be made incident on newly introduced state. The outgoing transitions from the newly introduced state will be made incident on the LCS.

4. Find out the end-point state of the if-else by finding out the common meeting state starting from the two transitions emanating from CSLCS, and doing a depth-first traversal. The common meeting state will be the end of the if-else condition starting at CSLCS. Let us call it Q_t. Let the cut-point next to Q_t be $q_{c_{tnext}}$.

5. Using chain copy mechanism described in earlier subsection, do a cut-point to cut-point copy of the contents along the two transitions emanating from CSLCS up to $q_{c_{tnext}}$ into M_s starting from the newly introduced state along its outgoing transitions as per the condition or its negation along the outgoing transitions. This will ensure the introduction of the entire if-else part on M_s, same as that starting at CSLCS. Go to *step 1*.

A formal definition of unmatched state is given below.

Definition 15 (Unmatched state (US) and its corresponding state)**.** *Let χ_u denote the mismatched path from LCS. Let $\chi_u^{(i)}$ be the longest prefix of χ_u such that it has matched* *with a prefix $\xi_s^{(j)}$ of some path ξ from CSLCS (in the golden FSMD M_g). The final state of the prefix $\chi_u^{(i)}$ is called the unmatched state (US). It may be noted that i may be zero, indicating that LCS itself is the unmatched state. The state in the FSMD of student's program M_s, which corresponds to US is referred to as the corresponding state of the unmatched state (CSUS).*

2.3.7 Simulation of correction mechanism on example problems

Example 2.5. The FSMDs and codes of teacher's and student's programs for calculating the tables for n numbers are reproduced below (Figures 2.66 and 2.67). The teacher's program for this problem requires two for loops. In student program the outer for loop is missing, making the student program inconsistent with teacher's program. We discuss below the simulation of the mechanism.

Table_correct.c

```
#include <stdio.h>
int main(){
    int n, i, num, j;
    printf("Enter an integer: ");
    scanf("%d",&n);
    for(j=0; j<n; j++){
        for(i=1; i<=10; ++i){
            num = n*i;
            printf("%d * %d = %d ", n, i, num);
       }
    }
    return 0;
}
```

Table_incorrect.c

```
#include <stdio.h>
int main(){
    int n, i, num, j;
    printf("Enter an integer: ");
    scanf("%d",&n);
    for(i=1; i<=10; ++i){
        num = n*i;
```

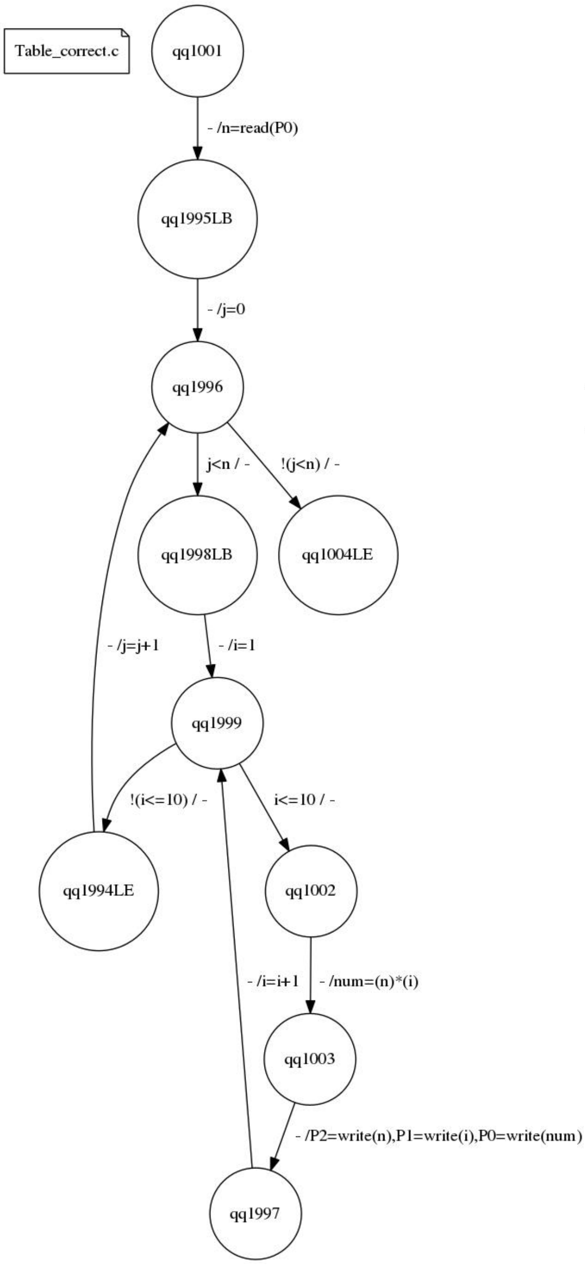


Figure 2.66: M_g, generate tables program

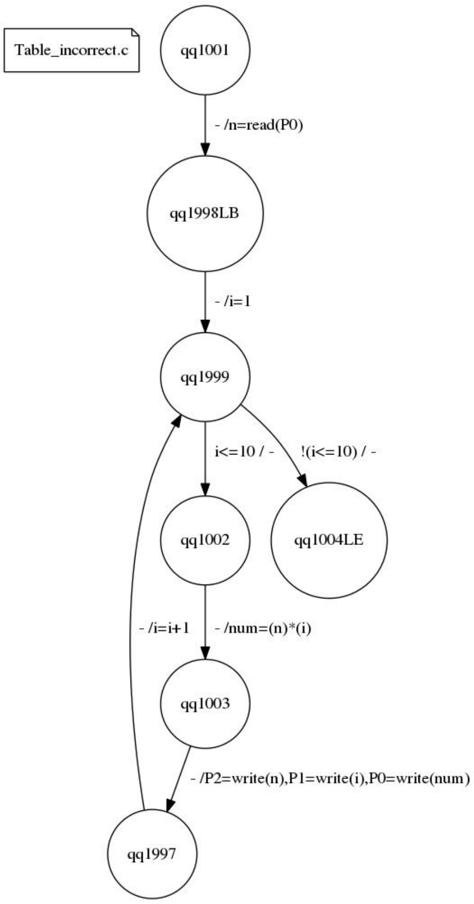


Figure 2.67: M_s, generate tables program

```
        printf("%d *%d = %d ", n, i, num);
    }
    return 0;
}
```

Execution of equivalence checker on these two FSMDs gives us $qq1001$ as the LCS. The CSLCS is the state $qq1001$. Containment checker takes the path $qq1001 \rightarrow qq1998LB \rightarrow qq1999$, starting from $qq1001$ to next cut point i.e $qq1999$ in M_s. It tries to find whether this path is present in M_g or not. Containment checker outputs the path $qq1001 \rightarrow qq1995LB \rightarrow qq1996 \rightarrow qq1998LB \rightarrow qq1999$ of M_g, which is containing this path. After the execution of step (3), we get the state $qq1998LB$ in M_s as *unmatched state*, which is also s_s. The corresponding state, CSUS, in M_g is $qq1995LB$, thus s_g is also $qq1995LB$ in M_g. DFS visit of M_g with $qq1995LB$ (i.e., s_g) as root gives that $qq1995LB$ is neither the starting of loop, nor this is a cut-point. Hence, ChainMg is constructed from s_g to the next cut-point $qq1996$, in M_g. ChainMs is constructed from US to the next cut-point $qq1999$, in M_s. The chain copy mechanism now works on M_s as follows. Both the chains are compared and found unequal by Equate_chains function. ChainMg is, therefore, introduced before US in step 6 (not shown, see figure 2.68). Now s_g moves down to the next state i.e., $qq1996$, in M_g. s_s remains at the same position, implying that the two FSMDs are the same upto s_s and s_g. Next, as s_g is a cut-point and a loop start state, so a self-loop loop1 is introduced before s_s in step (3) in M_s. The condition of the self-loop is made the same as the condition at s_g. Now s_g is moved down to the next state $qq1998LB$ in M_g. s_s remains at the same place implying that the two FSMDs are the same upto s_s and s_g. The FSMD M_s after this step is shown in figure 2.68. The unmatched state in this figure is numbered as $qq1998LELB$. The modified code of the student's program now becomes as follows.

```
#include <stdio.h>
int main(){
    int n, i, num, j;
    printf("Enter an integer: ");
    scanf("%d",&n);
    for(j=0; j<n; j++){
    }
        for(i=1; i<=10; ++i){
             num = n*i;
             printf("%d * %d = %d ", n, i, num);
        }
    return 0;
}
```

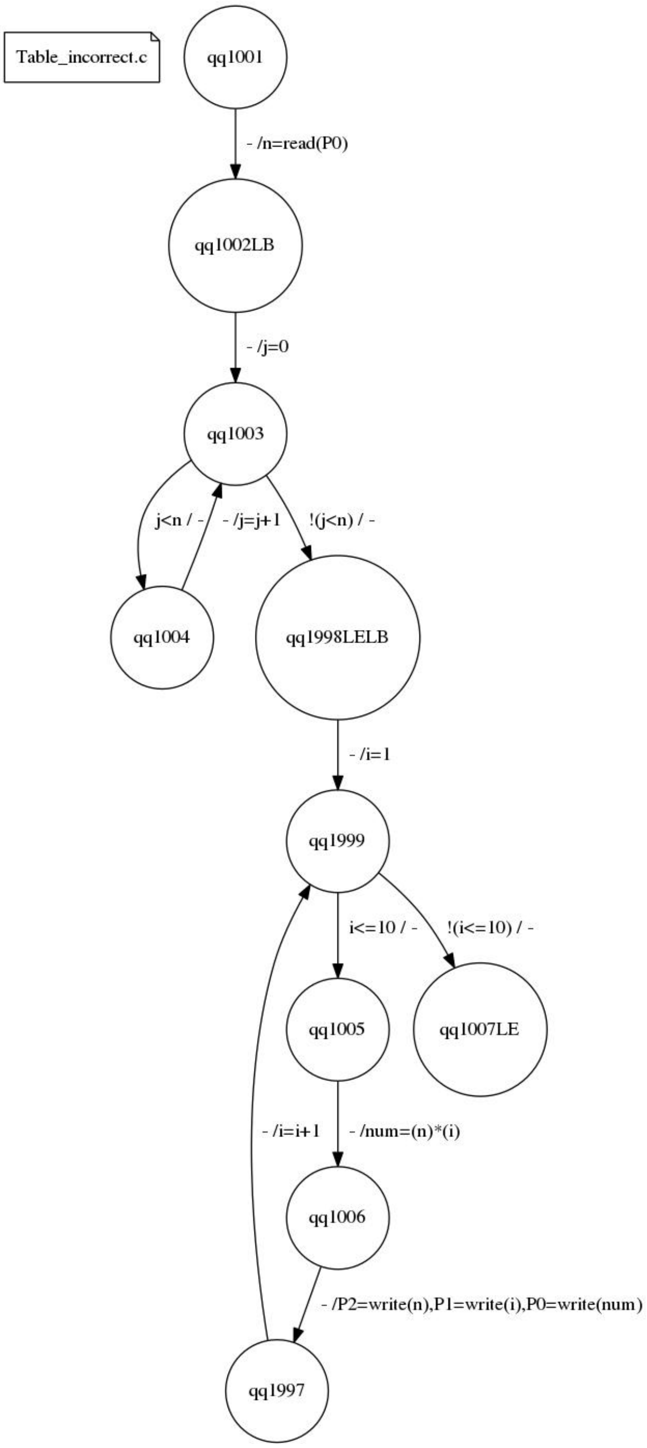


Figure 2.68: Modified incorrect FSMD after introducing loop1 before the unmatched state in M_s for generate tables program.

As the next statement after s_g is a single assignment leading to a cut-point, which is also the same as the next statement after s_s, so a trivial chain copy mechanism described earlier makes a copy of the assignment into the self-loop loop1. s_g now moves down to the next state qq1999 in M_g. Also the s_s moves down to the next state in M_s, implying that the two FSMDs are the same upto s_s and s_g. As s_g is now a loop start state, so a self-loop, loop2, is introduced inside loop1. The loop loop2 is shown introduced at the state qq1003 in the figure 2.69 for the modified FSMD M_s. The condition of loop2 is kept the same as the condition at s_g. Now s_g moves to the next state qq1002 in M_g. s_s also moves down to the next state, implying that the two FSMDs are the same up to s_s and s_g. The modified code of the student's program now becomes as follows.

```
#include <stdio.h>
int main(){
    int n, i, num, j;
    printf("Enter an integer: ");
    scanf("%d",&n);
    for(j=0; j<n; j++){
        for(i=1; i<=10; ++i){
        }
    }
    num = n*i;
    printf("%d * %d = %d ", n, i, num);
    return 0;
}
```

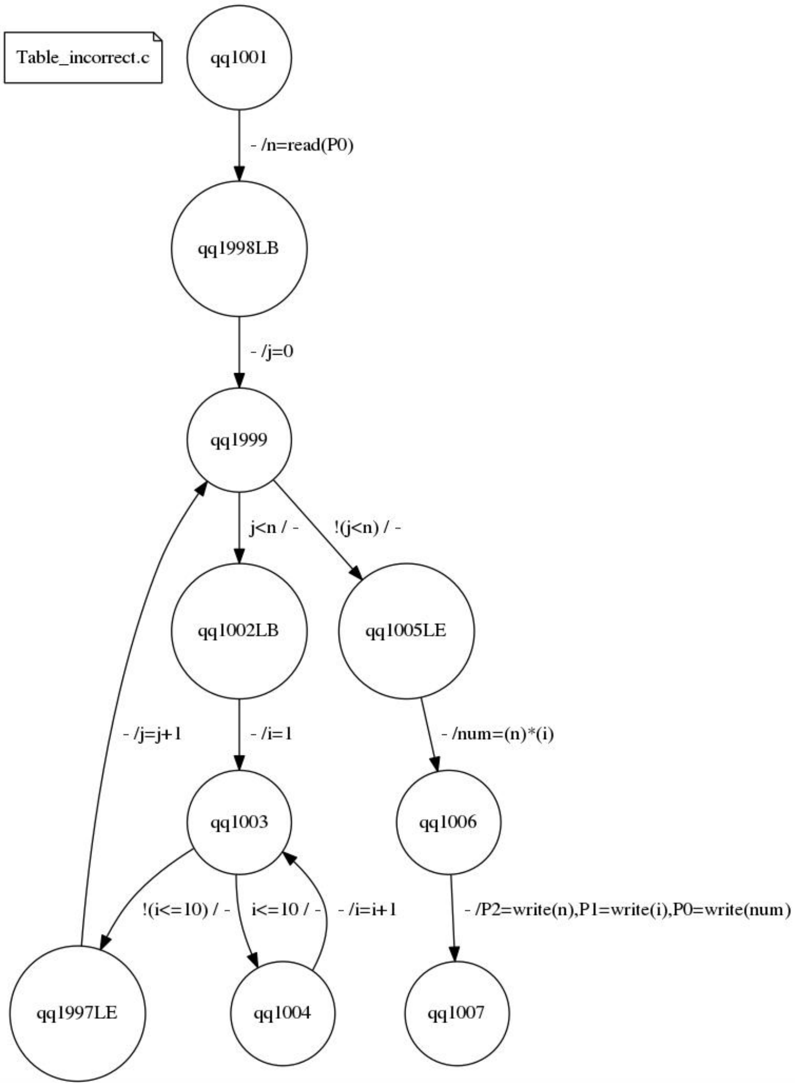


Figure 2.69: Modified incorrect FSMD after introducing loop2 inside loop1 in M_s for generate tables program.

Check the state s_g, as it is not a cut-point, so make a chain from s_g to the next cut-point. Also make a chain from s_s to the next cut-point. Using chain copy mechanism, copy the chain inside the loop loop2. This makes the FSMD M_s same as the FSMD M_g. The resulting M_s is shown in figure 2.70 and the resulting code of student's program is as shown below.

```
#include <stdio.h>
int main(){
    int n, i, num, j;
    printf("Enter an integer: ");
    scanf("%d",&n);
    for(j=0; j<n; j++){
        for(i=1; i<=10; ++i){
             num = n*i;
             printf("%d * %d = %d ", n, i, num);
        }
    }
    return 0;
}
```

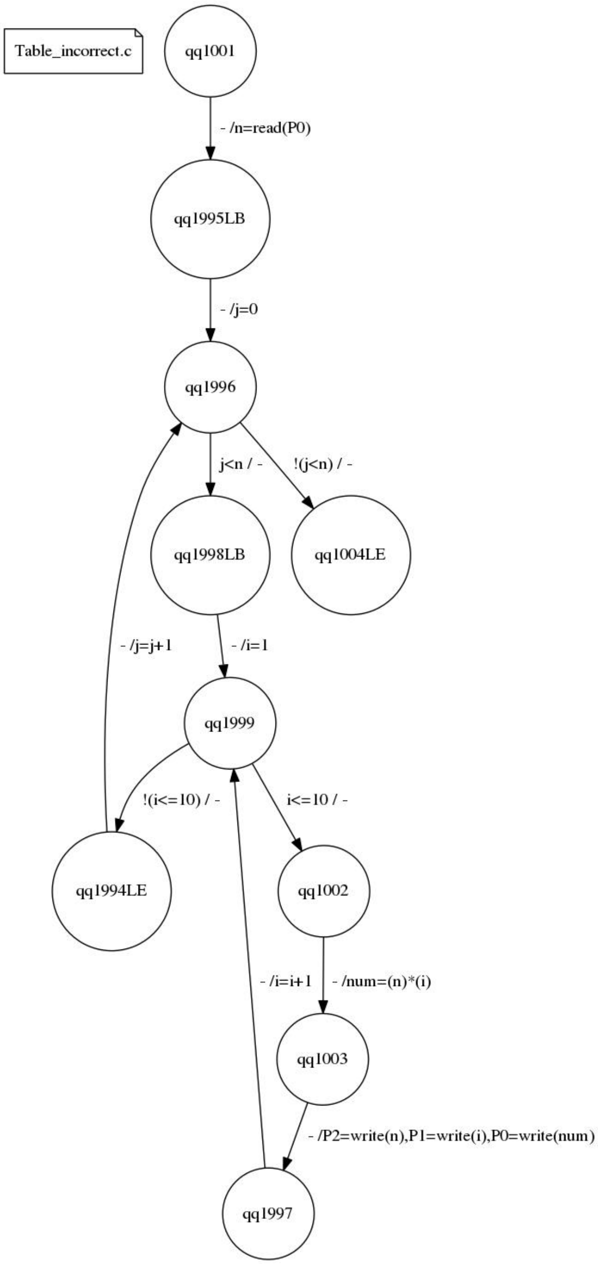


Figure 2.70: Modified incorrect FSMD after introducing chain in loop 2 in M_s for generate tables program.

2.3.8 Missing code of nested condition checking

In this case as the two FSMDs will not be equivalent, we verify this fact by checking the equivalence in both the directions. It is so because checking equivalence only in one direction may sometimes give misleading result. The LCS is obtained from the run of equivalence checker in which non-equivalence was reported. In this case the containment checking may give two types of results as given in section 2.3.3 for three cases of missing code. Both of the results of containment checking are treated by the following correction mechanism.

Given below are the two FSMDs (figures 2.71, 2.72). Assume the first FSMD as correct FSMD and second FSMD as incorrect FSMD. There are two nested conditions missing in the incorrect FSMD, which are nested inside another condition, in the correct FSMD. The portion of FSMD M_g between the states q_{06} to q_{10} , is missing in incorrect FSMD, M_g. This will make the equivalence checker fail.

Correction mechanism

The idea for finding the state from where we have to introduce the code is same as discussed in section 2.3.5. Introduce a node above LCS and add those transitions to newly introduced state which form the correction vector of corresponding state of LCS. The transitions previously incident on LCS will now be incident to newly introduced state. The outgoing transitions from the newly introduced state will now be incident on the LCS. We find out the end state of the if-else code, where the the branches of if and else meet. We make the outgoing branches from the new state have the same contents as the branches of if and else starting at CSLCS, using chain copy mechanism described earlier. Again we find the new LCS and CSLCS and keep on adding new states and copying the transitions until the correction vectors of CSLCS and transitions of LCS are not same. This will introduce the conditions and data transformations correctly in M_s.

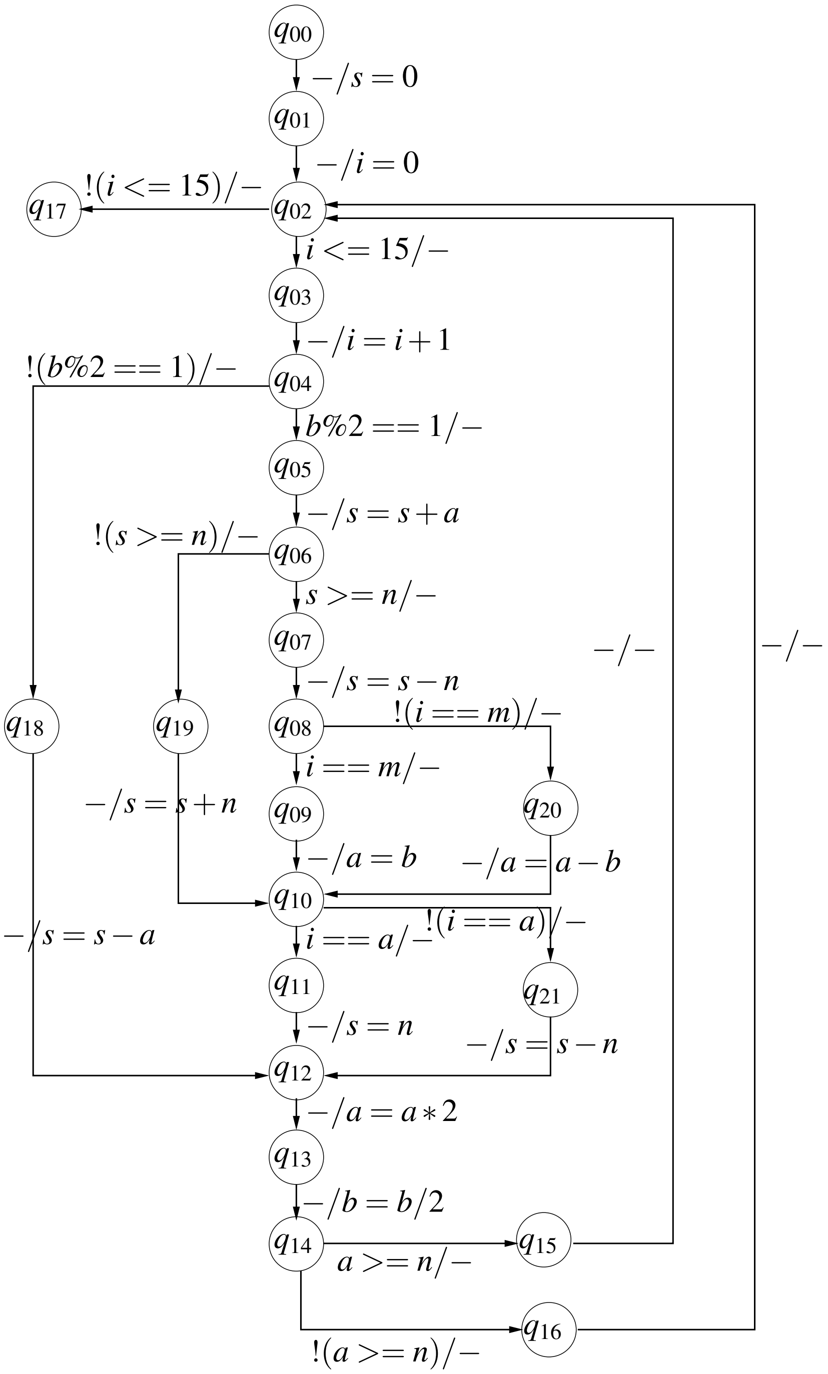


Figure 2.71: M_g having nested conditions.

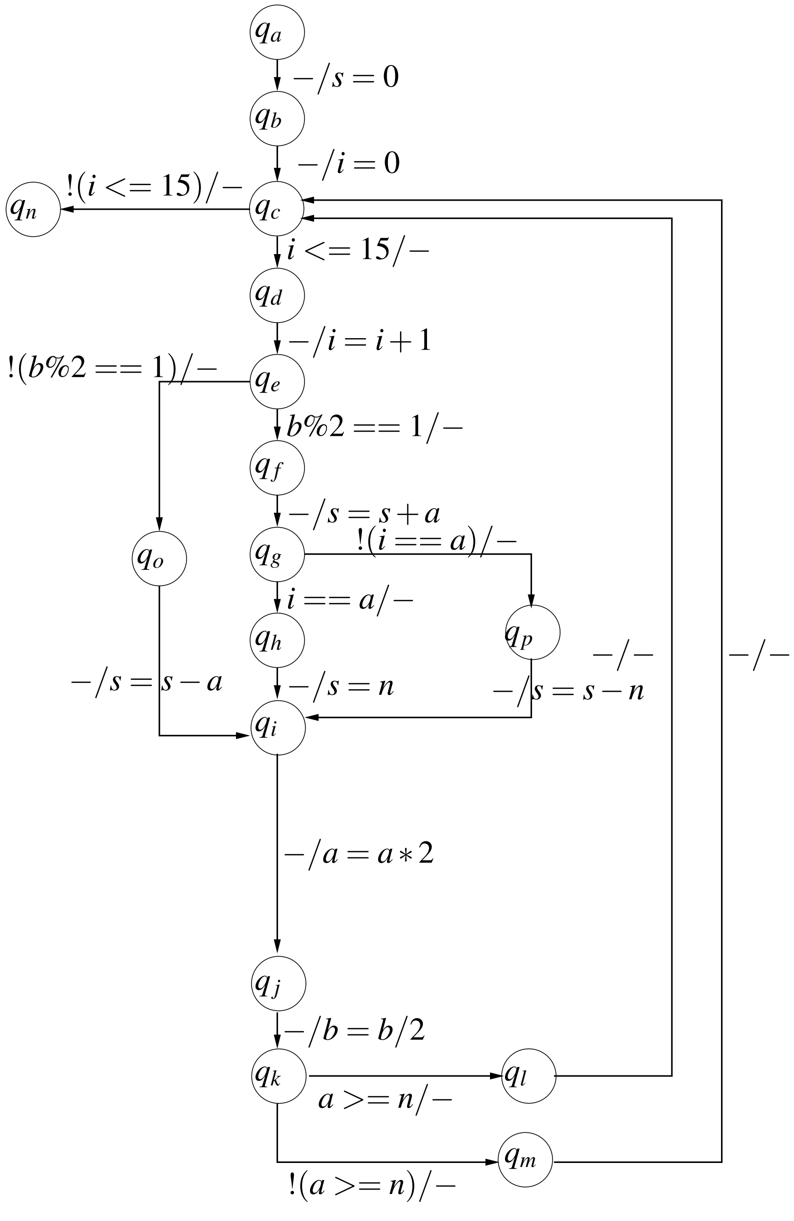


Figure 2.72: M_s missing nested conditions.

2.3.9 Steps for correction mechanism

Following steps will be executed after executing first four steps of correction mechanism in section 2.3.6

1. Find LCS and CSLCS. In case CSLCS is cut-point, not a loop start state, corresponding to if - else condition do the following.
2. If the correction vector of the CSLCS, C_v [CSLCS], is the same as the outgoing transitions from the LCS, then exit.
3. Introduce a state above LCS and make the outgoing transitions of the newly introduced state, the same as the ougoing transitions of CSLCS. The two transitions will correspond to a condition and its negation. The transitions previously incident on LCS will now be made incident on newly introduced state. The outgoing transitions from the newly introduced state will be made incident on the LCS.
4. Find out the end-point state of the if-else by finding out the common meeting state starting from the two transitions emanating from CSLCS, and doing a depth-first traversal. The common meeting state will be the end of the if-else condition starting at CSLCS. Let us call it Q_t. Let the cut-point next to Q_t be $q_{c_{tnext}}$.
5. Using chain copy mechanism described in earlier subsection, do a cut-point to cut-point copy of the contents along the two transitions emanating from CSLCS up to $q_{c_{tnext}}$ into M_s starting from the newly introduced state along its outgoing transitions as per the condition or its negation along the outgoing transitions. This will ensure the introduction of the entire if-else part on M_s, same as that starting at CSLCS. Go to *step 1.*

2.3.10 Simulation of above mechanism

We have simulated the above mechanism for two FSMDs. We are assuming the first FSMD as correct FSMD, M_g and second FSMD as incorrect FSMD, M_s. Incorrect FSMD is missing two nested conditions, which are nested inside another condition. The FSMDs M_g and M_s, are shown in figures 2.71 and 2.72. The portion of FSMD M_g between the states q_{06} to q_{10} , is missing in incorrect FSMD, M_g. This will make the equivalence checker fail.

From the execution of the equivalence checker on the above two FSMDs we find state q_g as LCS and q_{06} as the CSLCS. We introduce the new state q'_{06} in M_s, before LCS, and make the two outgoing transitions from it have the conditions same as the correction vector (outgoing transitions) of CSLCS. These outgoing transition are made incident on LCS. In the following figure 2.73, the new state and its outgoing transitions have been shown.

Now chain copy mechanism is invoked in the two outgoing branches from q'_{06}. This introduces the state q'_{19} and the subsequent transition in the branch corresponding to the negation of condition. This makes the entire branch copied. In the other branch, corresponding to the condition being true, the state q'_{07}, the subsequent transition and the next cut-point state q'_{08} is introduced. This is shown in the figure 2.74.

The chain copy mechanism is applied now in the outgoing branches from q'_{08}. As a result the states q'_{09} and q'_{20} are introduced and the relevant transitions are also introduced, giving the modified FSMD M_s, which is same as the golden FSMD M_g. Figure 2.75 shows the modified M_s as a result of these steps.

Summary

In this sub-section we have discussed the implementation of correction mechanism when one loop, out of two nested loops, is missing. We have simulated the correction mechanism on a student program to calculate sum of digits of input number and sum of digits of sum until a single digit number is obtained. We have also discussed the implementation of correction mechanism when nested conditions are missing. We have simulated the mechanism on an FSMD, in which incorrect FSMD is missing two nested conditions, which are nested inside another condition.

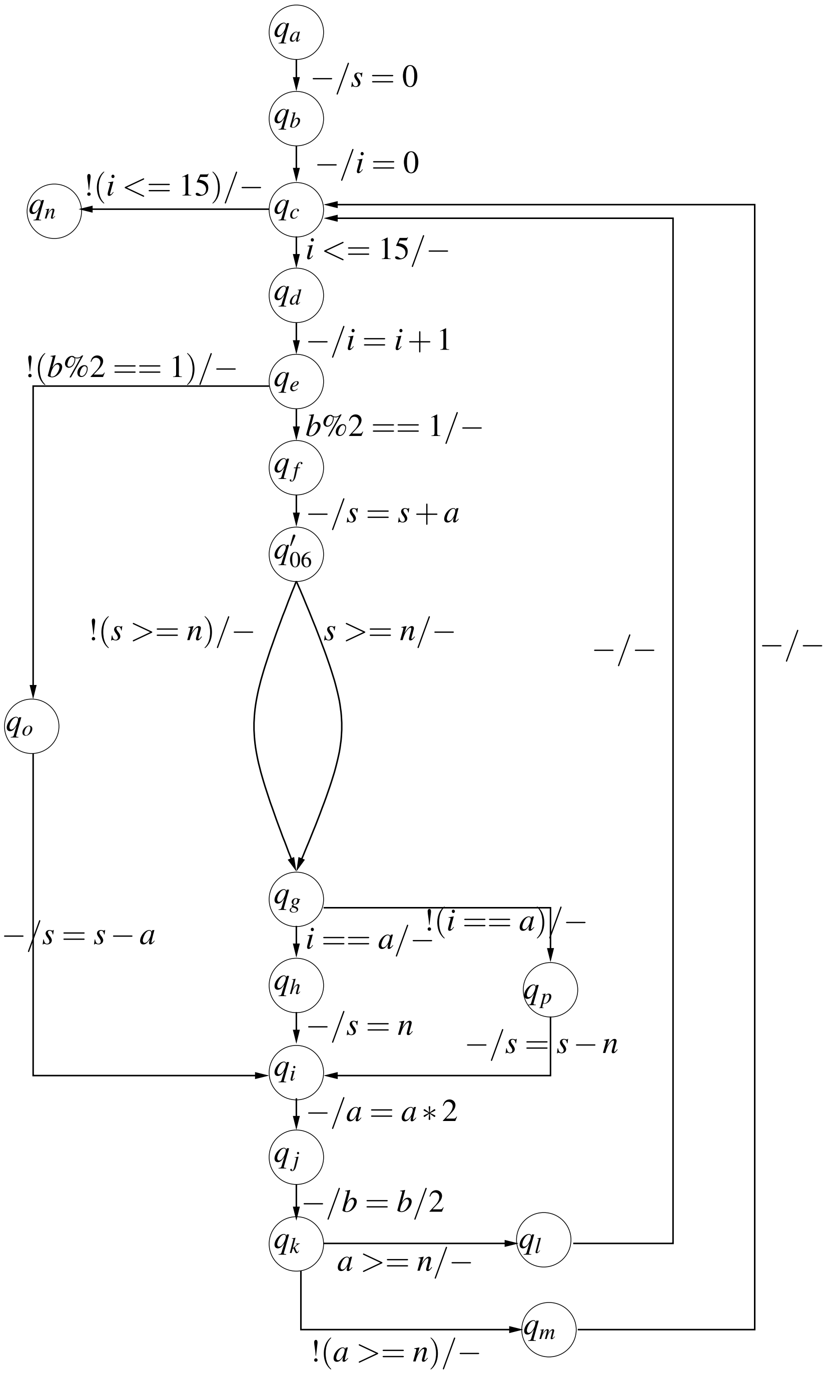


Figure 2.73: After introduction of missing state q'_{06} in M_s.

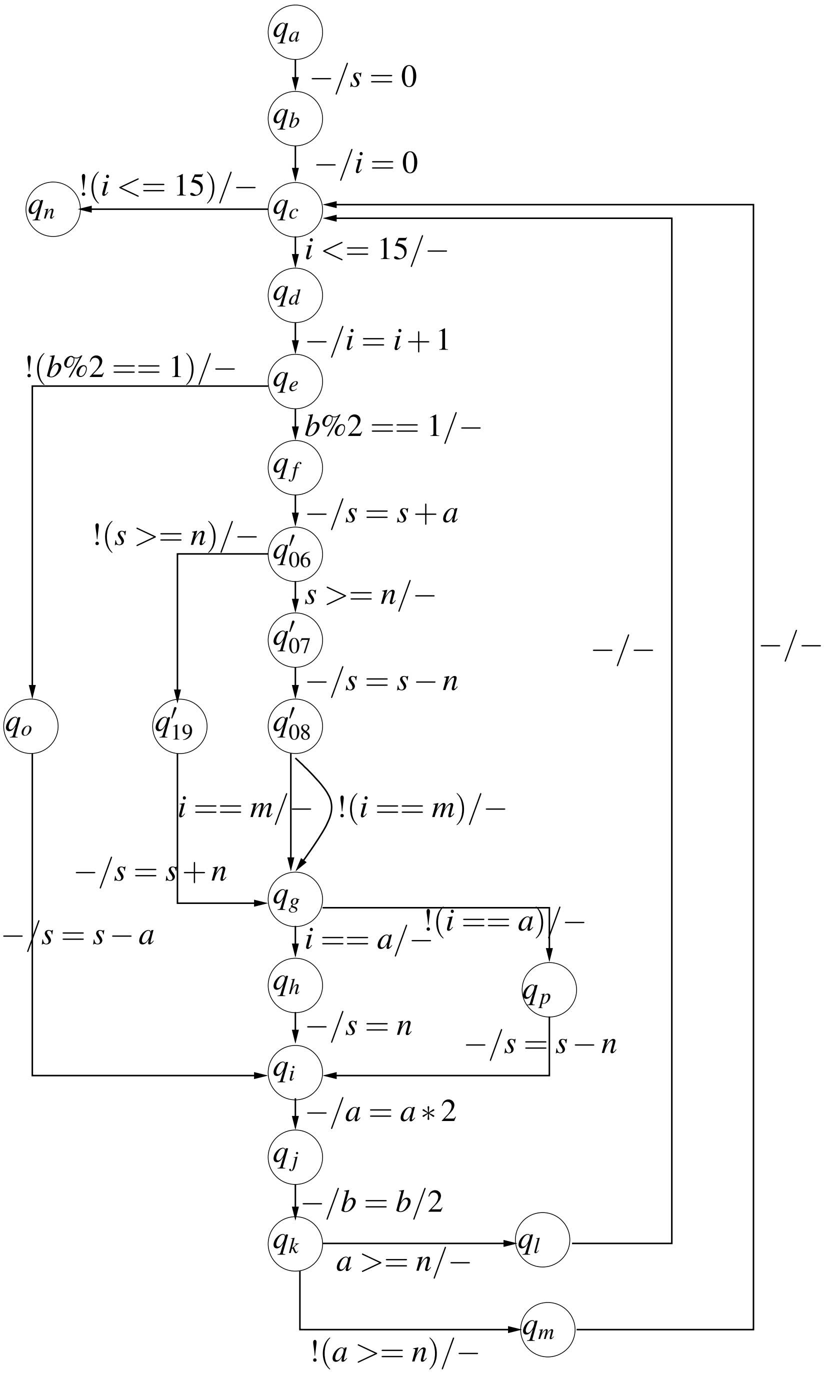


Figure 2.74: After introduction of missing states q'_{07}, q'_{08} and q'_{19} in M_s.

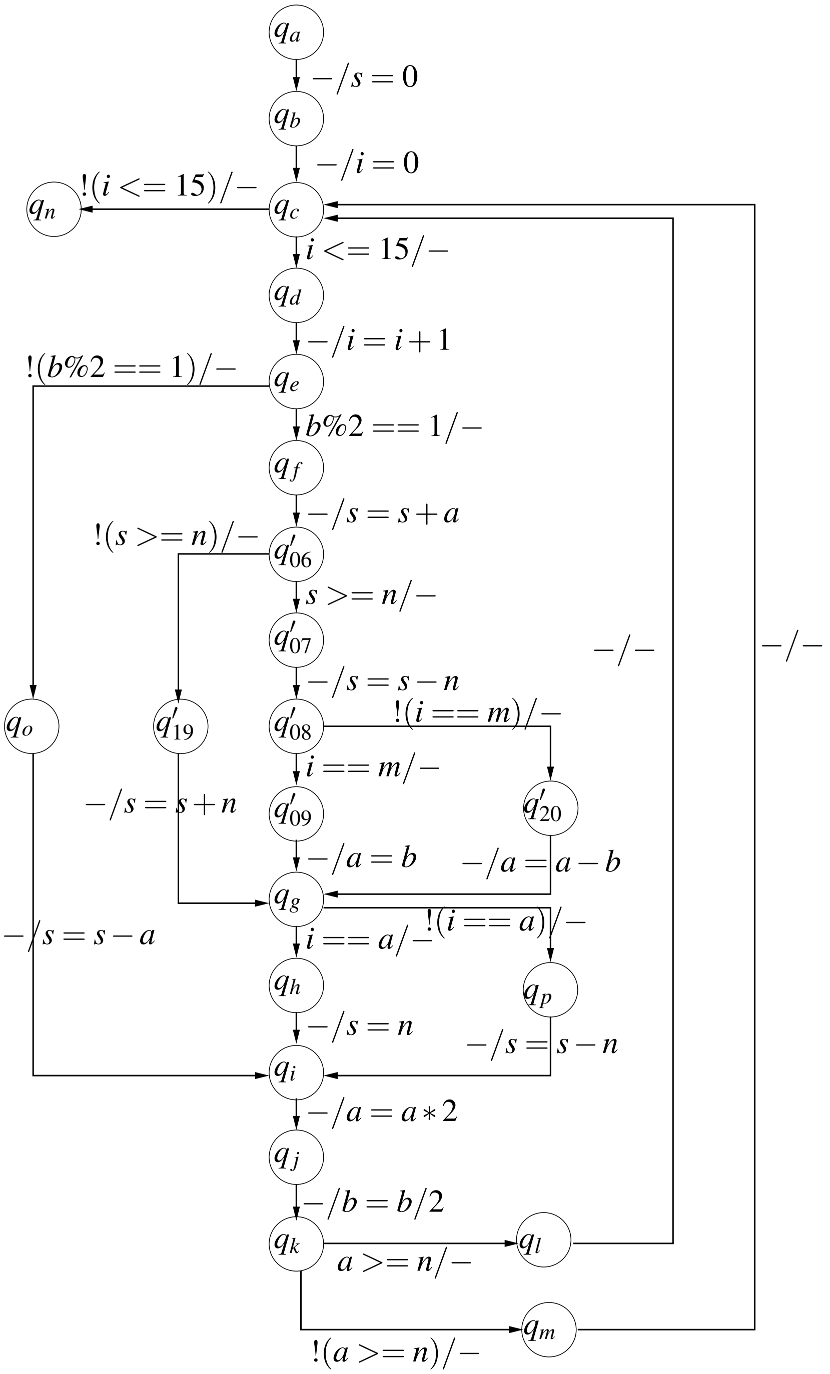


Figure 2.75: Corrected M_s, after introduction of missing states q'_{09} and q'_{20}.

2.4 Summary of the strategies applicable for all the cases discovered so far

We now summarize the algorithms for the correction strategies to be applied in order to do automated assessment of programs.

1. At first we discuss the algorithm for dependency violation. Dependency violation can be detected due to the fact that if the equivalence checker fails to find the equivalence between the incorrect and the correct FSMDs and a subsequent execution of containment checking mechanism reports *unordered pathwise both way contained*, then there must be a dependency violation inside the incorrect FSMD as per theorem 2. Failure of equivalence checker occurs as the equivalence checker cannot escape to account for a dependency violation as it incorporates forward substitution. In forward substitution the latter definition of a variable is substituted in place of the occurrence of variable subsequent to the new definition. This is why the equivalence checker is not able to detect the dependency violation. The containment checker is not sensitive to dependency violation, as it only tries to match the occurrence of a data-transformation among several of them, but does not bother in which order it encounters each data-transformation. For example, we see that dependency is violated in the following program segments. In the first one, which is a correct program, the expression `a = a + 1` follows the expression `s= s + a`. In `s = s + a`, the variable `s` is dependent on the variable `a`. In the second program segment, i.e., in the incorrect version, `a = a + 1` precedes `s = s + a`. Thus the variable `a` in `s = s + a` is dependent on `a = a + 1`. This causes the violation of dependency in the incorrect program. Due to forward substitution the equivalence checker converts the subsequent statement of the incorrect program into the following statement: `s = s + a + 1`. Thus the equivalence checker is able to differentiate between the values of `s` in the two programs, the correct one computes `s = s + a`, whereas the incorrect one computes `s = s + a + 1`. Thus equivalence checker will report the two programs as not equivalent. The containment checker, however, will only look for the existence of the two data transfer statements of the correct program, `a = a + 1` and `s = s + a` in the incorrect program and it finds them there, reporting *unordered path-wise both*

way contained.

2. We now discuss how a check for parenthesis skipping is done. This error arises if the student does not put the closing parenthesis at the end of a block, where it should be placed. Instead of this he puts it after some more code, which may belong to the next block of code. This will cause the corresponding edge to skip the state on which it should be incident in the FSMD of golden program. The skipped edge will thus be incident on some state beyond the original state, causing change in the FSMD structure.

   Parenthesis skipping is of following types *i)* parenthesis skipping to a cut point, *ii)* parenthesis skipping to a non cut point and *iii)* edge corresponding to skipped parenthesis belongs to a loop. Here we discuss the first case, as the algorithm for this case is applied to all incorrect programs for the reasons of automating the process. In this case, equivalence checking fails and the equivalence checker reports a path below LCS, from LCS to the next cut point, as the faulty path. This path as indicated by the equivalence checker, however, is not actually having the error. The error lies in the path above LCS, which is to be reported and the corresponding correct code needs to be informed to the user. The algorithm for this case employs pushing the set difference of number of incoming edges on the corresponding states of the FSMDs of the correct and the incorrect program on two stacks, stack1 and stack2. Stack1 is pushed with the set difference of incoming edges of a state on the FSMD of the correct program and the incoming edges of the corresponding state on the FSMD of the incorrect program, starting from a state and then going down to the next state below, till LCS. Stack2 is pushed with similar set difference of a state on the incorrect FSMD and the corresponding state on the correct FSMD. The state from which this mechanism is started is the least numbered state on the other ends of the edges incident on the LCS, as the edge for skipped parenthesis started from this state. We keep popping and moving to the states below till we find the top of the stacks have the same edge (i.e., same condition and data-transformation and the start states are also corresponding ones). This edge is popped from both the stacks. The states at the two ends of the popped edge in the incorrect program are the states between which there lies faulty code causing skipping of edges. The correct code is the code between the corresponding states in the correct program, which are respectively corresponding to the states at the two ends of the popped edge in the FSMD of the incorrect program. In actual algorithm, this mechanism is started at the start state of the program so that this algorithm can be applied in all cases, in order to automate the mechanism, as given a program we do not know which error will it have. It is to be noted that containment checker in this case will report *path-wise one way contained*. The containment checker also reports *path-wise one way contained* in case of missing code in the incorrect program. The algorithm for missing code is dealt with later in this section.

3. Next we discuss how we check for the error of parenthesis skipping, such that the skipping edge meets a non cut point state in the FSMD of incorrect program. In this case after the equivalence checker fails, the containment checker reports *partially equal and partially contained*. This step is, therefore, executed only if the containment checker reports *unordered path-wise both way contained and path-wise one way contained*. The error reporting mechanism is the same as that in the above case, but here the algorithm starts pushing into the stack starting from the state LCS on the FSMD of incorrect program and its corresponding state, CSLCS on the FSMD of correct program. The pushing continues with the subsequent states as in the previous step, continuing as before till the top of both the stacks is the same. Once the edge on the top is the same on both the stacks, it is popped and used to report the correct and incorrect code as in the previous step.

4. Edge corresponding to skipped parenthesis belongs to a loop.
   Because of the skipped parenthesis and resulting introduction of extra code inside the loop body, the start state of the loop in the correct FSMD will have two corresponding states in the incorrect FSMD. One of which is the start state of the for loop in the incorrect FSMD. The other corresponding state is the last correct state. The paths between LCS and start state of the loop is reported contained in the correct FSMD by the containment checker. The code corresponding to the paths between LCS and start state of the loop is suggested as extra code in the loop, this code should be moved away from the loop.

5. Missing code of nested condition checking - The unmatched state does not belong to loop.
   The term unmatched state is introduced in the discussion below. In this step we check for two cases of missing code of nested condition checking. *i)* In one case the equivalence checker gives a no answer and the containment checker reports

path-wise one way contained. ii) In the other case the equivalence checker when executed with incorrect FSMD as the first argument, shows up equivalence but when the arguments are reversed, it gives a no answer. The containment checker in this case reports *path-wise un- contained.* As the error reporting mechanism is the same in both the cases, so we proceed as follows. At first the equivalence checker is run with incorrect FSMD as the first argument. If the report is "not equivalent" then we proceed with the containment checker, which should report *path-wise one way contained*; but if it reports equivalent, then the arguments are reversed. If now the report is "not equivalent", then containment checker is run, which should report *path-wise un- contained.* If the containment checker output is as just mentioned for the two cases, then only we proceed with this mechanism as the other cases of containment checking have been handled in previous steps. The LCS lies on the correct FSMD in the case where we had to reverse the arguments for equivalence checking. The mechanism in these cases is basically to find out the path starting from CSLCS, which contains the path from LCS to next cut point. Comparing cut point to cut point we find out the unmatched state on the path from LCS. Unmatched state is the state up to where the FSMDs match and after that code is missing. To insert the missing code, correction vector is used and new states are introduced and list of corresponding states is updated until the correction vectors of the corresponding states match. In order to ensure correct nesting the method described earlier with two stacks is used.

6. Missing code belongs to loop - The Unmatched State belongs to loop.
First, the unmatched state (US) and the corresponding state of unmatched state (CSUS) are found in a manner similar to the previous step. The mechanism in these cases is basically to find out the path starting from CSLCS, which contains the path from LCS to next cut point. Comparing cut point to cut point, we find out the US on the path from LCS. CSUS is then found from the correct FSMD. It is then checked whether CSUS is the starting state of a loop, in which case, a new state is introduced in the incorrect FSMD before the unmatched state and a loop is introduced with the newly introduced state as the start state of the loop. Actually, the correction vector (the only outgoing edge, in this case) of the start state of the loop in the correct FSMD (which is the CSUS also) is added to the newly introduced state, as the loop-back edge. Code is added in the incorrect FSMD, from the loop previously present in the incorrect FSMD to the newly

introduced loop, until it is found that the the feedback coming to the newly introduced state has become equal to the feedback coming to CSUS, the starting state of loop in the correct FSMD. For checking, this it is to be ascertained that the loop-back edge is coming from the corresponding states in both the FSMDs and the condition and data transfer are the same on the loop-back edges in both the FSMDs. Further, we intend to push the entire remaining code of the already existing loop to the newly introduced loop. For this, single statements are pushed one after the other, and a block of code (e.g. loop, conditions or nested conditions etc.) is pushed in its entirety at one time. Every time we push some code, we check whether the feedback edge has become equivalent to the feedback edge in the correct FSMD. After pushing the entire already existing loop's code to the newly introduced loop may give us the situation that the state from which feedback is coming to the newly introduced state is the corresponding state of the state from which feedback is coming in the correct FSMD, condition of the feedback is also the same but data-transformation may be different. This may further require pushing the next data-transformation or a block of code (which lies outside the previously existing loop) inside the newly introduced loop. This will give us the FSMD which is the same as the correct FSMD.

2.5 Results and discussions

The following table 2.1 shows the results of containment analysis for various cases identified for 10 samples of each case. In most of the cases, the error has been detected correctly. Table 2.1 thus summarizes the various error cases handled so far and the reports of equivalence checker and the containment checker for each case. The reports of equivalence checker and the containment checker are same for cases 2,4,5 and 7 in the table. Case 2 occurs when the parenthesis is misplaced in such a way that the corresponding branch in FSMD becomes incident on a cut-point state, which lies beyond the actual state on which it should have been incident. Case 4 occurs when due to wrongly putting the closing parenthesis of a loop body, the code beyond the loop body also becomes a part of loop. Cases 5 and 7 are under the category of error of missing code. Case 5 occurs when there is missing code causing a non-nested cut-point missing in the FSMD. Case 7 is when there are two nested cut-points missing.

In all these cases, the containment is found after doing path extension, as in the case of *path-wise one way contained.* These cases can be distinguished based on the facts that *i)* in case of faulty branching due to misplaced closing parenthesis, at least two states will have different number of incident branches when compared with the golden FSMD and *ii)* in case of faulty branching from inside the loop body to some state outside the loop, resulting in extra code inside loop, there is always a loop and the start state of the loop gets two corresponding states in the incorrect FSMD. The case of missing code will not have these two situations. The three cases are thus separable.

In this chapter the cases missing nested loops and missing nested conditions were also aimed at. Schemes were suggested and have been found to work on the example problems as summarized in table 2.2. The results of our study in the table 2.2 suggest that in case the student's program has code which is in order with the golden program, then the correct code can be recovered and no duplicate code will be inserted in the recovered code. Some of the example codes mentioned in table 2.2 are are given in the appendix B.

We can thus summarize the result of our work in this chapter as follows.

1. The equivalence of programs can be effectively checked using an additional containment checking approach. Incorporation of containment checking is required for identifying the type of of errors, e.g., dependency violation etc. as in the table 2.1.

2. Program errors which lead to alteration in the FSMD structure may be classified broadly into various categories according to the resulting FSMD structure as outlined in item 2 of the work done section above and the ones that we studied are given in the table 2.1.

3. Strategies were developed for diagnosing errors as above and for reporting the faulty portion of the code. Some results are in table 2.1, from which it is evident that error diagnosis is being correctly done in most of the program FSMDs.

Case	Error type	Equivalence checker output	Containment checker output	#tested	#detected
1	Dependency violation	Not contained	Unordered path-wise both way contained	10	7
2	Parenthesis skipping to cut point	Not contained	Path-wise one way contained	10	10
3	Parenthesis skipping to non cut point	Not contained	Unordered path-wise both way contained and path-wise one way contained for faulty branching	10	10
4	Skipping parenthesis belonging to a loop	Not contained	Path-wise one way contained	10	7
5	Code missing (missing non-nested cut-point)	Not contained	Path-wise one way contained	10	10
6	Code missing (missing cut-point is nested inside another cut point)	contained, not contained when reversed	Path-wise un-contained	10	10
7	Code missing (two nested cut points are missing)	Not Equivalent	Path-wise one way contained	10	10

Table 2.1: Table of cases of error diagnosis of program FSMDs

Program	Error type	Nature of code in M_s	Applicability of scheme	Result
Tables	missing outer loop	in-order	yes	restores missing loop, no duplicate code resulted
Conditions	missing nested conditions	in-order	yes	restores missing conditions, no duplicate code resulted
Digitsum	missing outer loop	in-order	yes	restores missing loop, no duplicate code resulted
Digitsum	missing inner loop	in-order	yes	restores missing loop, no duplicate code resulted
Diamonds	missing both outer, inner loops	in-order	yes	restores missing loops, no duplicate code resulted
Simple	missing some transitions	in-order	yes	restores missing transitions in order, no duplicate code resulted

Table 2.2: Table of cases of error handling of program FSMDs for missing nested or in-line code

2.6 Conclusion

In this chapter we have modified the equivalence checking by adding containment checking to it in order to be able to detect faults in the programs while checking their equivalence with model programs. Four cases of containment types have been checked and the results have been found to be satisfactory in the sense that we have been able to find out the fault and report remedial suggestions for them. In future we propose to enhance the work by incorporating error detection for multiple errors and error correction strategies.

This chapter assumed firstly that the FSMD of student's program has the conditional constructs in the proper order, thus there is no dead code in the student's program due to improper ordering of conditional constructs. The second assumption was that the FSMDs being compared have the same variable names. The two assumptions are not true for actual student's programs. We describe our methods to handle the cases, where the student's programs may have the above two deficiencies, in the next chapter.

Chapter 3

Methods to reconcile dissimilarities between FSMDs arising from students' programs

3.1 Introduction

In this part of the work, pre-processing requirements of programs have been aimed at. The research objectives for this part of the work were identified as (i) developing methods to support automated evaluation, in cases where programs have conditional constructs, which should obey precedences and (ii) to develop a scheme for variable mapping in programs. Work done in this chapter is introduced below.

The first aspect mentioned above is due to the fact that logic in programs demands that in a nesting of if statements there are conditions that have to be evaluated in a certain order, but the students may violate that order. Before thorough equivalence checking is done, the student's program, therefore, has to be subjected to a pre-processing step. The objective of pre-processing is that a mapping of the names of variables have to be evolved and that the nested conditions should conform to some rule of precedence. In the following section we first describe variable mapping and then the problem of identifying the precedence order.

The second aspect is due to the fact that since the students will be using variable names different from those in the golden program, we will have to evolve a mapping or an association of the variables used in the student's program, with the ones used in the golden model. An algorithm has been developed for variable mapping between two programs. This is done as the FSMD based equivalence checking assumes the variable names to be the same in the two programs under examination, without which the equivalence checking is not possible. The variable mapping algorithm is an FSMD driven algorithm in the sense that it prepares the FSMD models of both the programs, compares their paths for similarity of conditions and data-transformation in a depth first manner and tries to establish a mapping between the variables, which assume equivalent symbolic values after traversing a path from a cut point to the next. Presently variable mapping is done as a pre-processing step. This may be enhanced in the future to work hand-in-hand with the equivalence checking steps.

3.2 Programs with constraints in ordering of conditions

A construct that appears frequently in programs is the `else-if` block. The condition that appears in a subsequent `else-if` block should not be stronger (i.e., less general) than any of its preceding conditions, otherwise this block will never be executed (in other words, this block qualifies as dead code). Such wrong ordering of conditions in `else-if` blocks are often introduced by novice student programmers. It is to be noted that such violation of precedence of conditions in `else-if` constructs qualify as logical errors which are not detected by standard compilers, such as *gcc* [2]. In this work, we tabulate different cases to identify the correct and the incorrect precedence of conditions in `else-if` constructs and thereby, automatically report such errors in students' programs and also provide feedback towards error correction.

Recently, an automated evaluation scheme based on equivalence checking of the finite state machine with data-path (FSMD) model [33] has been proposed in [77]. Specifically, FSMD based equivalence checking was first proposed in [45], which is later developed to handle uniform and non-uniform code motion based optimization techniques in [44, 47, 54]. This method is general enough for checking equivalence of digital circuits as well [46]. A further enhancement of this method can be found in [11] and [12], which can additionally handle code motions across loops. Thus, by adopting the FSMD based equivalence checking method, the method of [77] can handle a wide range of supported code optimization techniques that may be applied by a student.

The method of [77] relies on the resemblance of the control flow of the student program with that of the golden model program supplied by the instructor/teacher. Note that there may be many ways to correctly order the conditions of an `else-if` construct and a student should be awarded full marks for following any one of these correct orderings; however, to achieve the same, the method of [77] needs to be supplied with all the possible correct permutations of the conditions of the `else-if` construct by the instructor. Thus, the instructor, theoretically, may need to supply an exponential number of golden programs (in terms of the number of conditions in an `else-if` construct) and hence the method of [77] is not viable. Therefore, we tabulate different cases to identify the correct and the incorrect precedence of conditions in `else-if` constructs and thereby, automatically report such errors in students' programs and also provide feedback towards error correction.

The following description is organized as follows. Section 3.2.1 describes the problem in detail and explains the basic technique for identification and correction of precedence of conditions in `else-if` constructs in students' programs. Section 3.2.2 describes the basic algorithm of our automated checking method; a formal treatment of the worst case time complexity of our algorithm is also provided in this section.

3.2.1 Identification and correction of precedence of conditions in `else-if` constructs

Algorithms with nested conditional branches are generally implemented using the `else-if` constructs in high-level languages, such as C and Java. The conditions that appear in the `else-if` blocks need to be in a well defined order; specifically, the condition that appears in a subsequent `else-if` block should not be stronger than any of its preceding conditions. Violation of such ordering occurs when a programmer wrongly places a condition, which should be checked in a latter block, in an earlier block – such cases are often encountered while checking novice students' programs. These errors are adjudged as logical errors which are not checked by the standard compilers, such as *gcc*. Hence, while checking the students' programs, we need to carefully examine that in the `else-if` blocks, the conditions have been placed in the proper order. If a condition which should be checked latter is kept in an earlier block and the condition which was to be checked earlier is put in a latter block, then at the time of execution of the program, the first placed condition will be checked and the corresponding code shall be executed but the latter placed condition will never be checked and thus the corresponding code-block will never get executed. Thus, we arrive at a piece of code which is analogous to *dead-code*. The reason for this is the fact that the test for every condition corresponds to a set of values of the variables in question and if this set, as determined by the first condition, happens to be a superset of the set corresponding to the subsequent condition, then the subsequent condition will never be checked because a superset of the condition has already been checked. As an example, we consider the following code:

```
if (x > 30)
    sum = sum + 1;
else if (x > 25)
    sum = sum + 2;
else if (x > 20)
    sum = sum + 3;
else if (x > 15)
    sum = sum + 4;
else
    sum = sum + x;
```

In the above code all the `else-if` blocks get a chance to get executed for some value of $\texttt{x} > \texttt{15}$. If $\texttt{x} \leq \texttt{15}$ then the else part is executed.
Now, let us take a look at the following code:

```
if (x > 15)
    sum = sum + 1;
else if (x > 20)
    sum = sum + 2;
else if (x > 25)
    sum = sum + 3;
else if (x > 30)
    sum = sum + 4;
else
    sum = sum + x;
```

In the above code if x $>$ 15 then the first block is executed, otherwise the else part is executed. For no value of x, the blocks corresponding to the conditions x $>$ 20, or x $>$ 25, or x $>$ 30 ever get executed. The reason here is violation of the ordering of the conditions.

From the discussion above, it is evident that the ordering of the test conditions in a code with else-if statement has to follow some rules of precedence; we have tabulated different such cases as shown in Table 3.1. Note that the column "Precondition" in this table enlists the relationship between the participating conditions in the else-if blocks; the entry $c_1 \Rightarrow c_2$ in this column indicates that c_1 is a stronger condition than c_2, i.e., whenever c_1 is satisfied, c_2 must also be satisfied. It is important to note that although we show at most two conjuncts (e.g., $c_1 \&\& c_2$) and two disjuncts (e.g., $c_1 \mid\mid c_2$) in Table 3.1.

The entries in Table 3.1 can be understood as follows. Given a precondition, if the program has code in the form depicted under the column for the "incorrect sequence", then the correct version would be as is written under the column "correct sequence". For example, in the case 1 as $c_1 \Rightarrow c_2$, hence check for c_1 should not be used after the check for c_2 in an if-else if block, as c_1 being a stronger condition, the else if part will never be executed, when both c_1 and c_2 hold, because the execution of if (c_2) will result in the execution of if part. The rules in Table 3.1 are not comprehensive. In cases, where none of the rules hold, our system will not be able to give any indication.

The rules of table 3.1 may be extended through careful analysis. Some examples of extension have been shown in the table 3.2. However, if two conditions are such that there is no containment relationship between them, then the ordering of these conditions is inconsequential. For example, if the conditions have different independent variables, then there cannot be any containment relationship.

Steps of the procedure automated precedence checker - for checking violation of precedence of conditions

For any nested if-else if statement *s*

Step 1: Let $\{C_1, C_2, \ldots C_k\}$ be the conditions associated with each if-clause in the order in which they occur in s;

Table 3.1: Examples for `else-if` constructs

Case	Precondition	Correct sequence	Possible Incorrect sequence

1	$c_1 \Rightarrow c_2$	if (c_1) {...}
		else if (c_2) {...}
			if (c_2) {...}
			else if (c_1) {...}

2	$c_3 \Rightarrow c_1$ or $c_3 \Rightarrow c_2$	if (c_3) {...}
		else if ($c_1 \parallel c_2$) {...}
			if ($c_1 \parallel c_2$) {...}
			else if (c_3) {...}

3	$c_1 \Rightarrow c_3$ or $c_2 \Rightarrow c_3$	if ($c_1 \&\& c_2$) {...}
		else if (c_3) {...}
			if (c_3) {...}
			else if ($c_1 \&\& c_2$) {...}

4	$c_1 \Rightarrow c_3$ and $c_2 \Rightarrow c_4$	if ($c_1 \&\& c_2$) {...}
		else if ($c_3 \parallel c_4$) {...}
			if ($c_3 \parallel c_4$) {...}
			else if ($c_1 \&\& c_2$){...}

5	$c_1 \Rightarrow c_3$ and $c_2 \Rightarrow c_4$	if ($c_1 \parallel c_2$) {...}
		else if ($c_3 \parallel c_4$) {...}
			if ($c_3 \parallel c_4$) {...}
			else if ($c_1 \parallel c_2$) {...}

6	$c_1 \Rightarrow c_3$ and $c_2 \Rightarrow c_4$	if ($c_1 \&\& c_2$) {...}
		else if ($c_3 \&\& c_4$) {...}
			if ($c_3 \&\& c_4$) {...}
			else if ($c_1 \&\& c_2$) {...}

Step 2: For any $i, 1 \leq i < k$, for any $j > i$, if $C_j \Rightarrow C_i$ then there is a violation of precedence.

3.2.2 Implementation

In this section, we present the procedure that we have devised to detect violation of precedence of conditions in `else-if` constructs of student's programs. The procedure consists of steps as elaborated below; we provide relevant examples in each step for clarification.

Step 1: If there are nested `else-if` constructs in a program, then we extract the conditions from the innermost `else-if` block and gradually move outwards; for example, consider the following program snippet:

```
if (c1) { ... }
else if (c2)
```

Table 3.2: Extended examples for `else-if` constructs

Case	Precondition	Correct sequence	Possible Incorrect sequence

1	$c_1 \Rightarrow c_2 \Rightarrow \ldots \Rightarrow c_n$	if (c_1) {...}
		else if (c_2) {...}
		$\ldots$
		else if (c_n) {...}
			if (c_n) {...}
			else if (c_{n-1}) {...}
			$\ldots$
			else if (c_1) {...}

2	$c_n \Rightarrow c_1$ or $c_n \Rightarrow c_2$ or
	$\ldots$ or $c_n \Rightarrow c_{n-1}$
		if (c_n) {...}
		else if $(c_1 \Vert c_2 \Vert \ldots \Vert c_{n-1})$ {...}
			if $(c_1 \Vert c_2 \Vert \ldots \Vert c_{n-1})$ {...}
			else if (c_n) {...}

3	$c_1 \Rightarrow c_n$ or $c_2 \Rightarrow c_n$ or
	$\ldots$ or $c_{n-1} \Rightarrow c_n$
		if $(c_1\&\&c_2\&\&\ldots\&\&c_{n-1})$ {...}
		else if (c_n) {...}
			if (c_n) {...}
			else if $(c_1\&\&c_2\&\&\ldots\&\&c_{n-1})$ {...}

4	$c_1 \Rightarrow c_{n+1}$ and $c_2 \Rightarrow c_{n+2}$ and
	$\ldots$ and $c_n \Rightarrow c_{2n}$
		if $(c_1\&\&c_2\&\&\ldots\&\&c_n)$ {...}
		else if $(c_{n+1} \Vert c_{n+2} \Vert \ldots \Vert c_{2n})$ {...}
			if $(c_{n+1} \Vert c_{n+2} \Vert \ldots \Vert c_{2n})$ {...}
			else if $(c_1\&\&c_2\&\&\ldots\&\&c_n)${...}

5	$c_1 \Rightarrow c_{n+1}$ and $c_2 \Rightarrow c_{n+2}$ and
	$\ldots$ and $c_n \Rightarrow c_{2n}$
		if $(c_1 \Vert c_2 \Vert \ldots \Vert c_n)$ {...}
		else if $(c_{n+1} \Vert c_{n+2} \Vert \ldots \Vert c_{2n})$ {...}
			if $(c_{n+1} \Vert c_{n+2} \Vert \ldots \Vert c_{2n})$ {...}
			else if $(c_1 \Vert c_2 \Vert \ldots \Vert c_n)$ {...}

6	$c_1 \Rightarrow c_{n+1}$ and $c_2 \Rightarrow c_{n+2}$ and
	$\ldots$ and $c_n \Rightarrow c_{2n}$
		if $(c_1\&\&c_2\&\&\ldots\&\&c_n)$ {...}
		else if $(c_{n+1}\&\&c_{n+2}\&\&\ldots\&\&c_{2n})$ {...}
			if $(c_{n+1}\&\&c_{n+2}\&\&\ldots\&\&c_{2n})$ {...}
			else if $(c_1\&\&c_2\&\&\ldots\&\&c_n)$ {...}

```
{
   if (c_1) { ... }
   else if (c_2) { ... }
   ...
   ...
   else { ... }
}
else if (c3) { ... }
...
...
else { ... }
```

For this program snippet, we shall first resolve the precedence of conditions of the inner `else-if` construct (comprising `c_1`, `c_2`, etc.) before resolving those of the outer `else-if` construct (comprising `c1`, `c2`, `c3`, etc.). Note that choosing the outer `else-if` construct first and then the inner one would have resulted in the same output; however, since a *mechanized* algorithm has to follow a specific order, we have decided to resolve the inner construct earlier.

Step 2: In order to compare two conditions, one needs to convert these conditions to normalized expressions first which makes them amenable to application of symbolic reasoning; this step explains how we convert a condition in an `else-if` construct to its corresponding normalized form.
(a) Each condition is parsed by delimiters “&&”, “||”, “(” and “)”.
(b) If the condition contains nesting of &&’s and ||’s, then we distribute ||’s over &&’s to convert the condition into conjunctive normal form, for example, $c_1 \mid\mid (c_2 \ \&\& \ c_3)$ will be converted into $(c_1 \mid\mid c_2) \ \&\& \ (c_1 \mid\mid c_3)$.
(c) Each arithmetic clause (i.e., clause without any && or ||) is converted to the form: *expression* $\langle$*relational operator*$\rangle$ *zero*, eg., `x == y` is converted into `x - y == 0`. Note that the expressions on the left hand side of the relational operator are normalized following the normalization technique described in detail in [47, 75]; here we enlist the rules of the normalization grammar and explain them briefly.
Normalization grammar:

1. $S \rightarrow S+T \,|\, c_s$, where c_s is an integer.

2. $T \rightarrow T * P \big| c_t$, where c_t is an integer.
3. $P \rightarrow \text{abs}(S) \big| (S) \,\text{mod}(C_d) \big| f(\mathit{list}\ S) \big| S \div C_d \big| v \big| c_m$, where v is a variable, and c_m is an integer.
4. $C_d \rightarrow S \div C_d \big| (S) \,\text{mod}(C_d) \big| S$,
5. $\mathit{list}\ S \rightarrow \mathit{list}\ S, S \big| S$.

In the above grammar, the nonterminals S, T, P stand for (normalized) sums, terms and primaries, respectively, and C_d is a divisor primary. The terminals are the variables belonging to the set of input and storage variables, the interpreted function constants abs, mod and $\div$ and the user defined uninterpreted function constants f. An example of user defined uninterpreted function constant is $f(v_1, v_2, 4)$, which will be normalized as $f(1 * v_1 + 0, 1 * v_2 + 0, 4)$. In addition to the syntactic structure, all expressions are ordered as follows: any normalized sum is arranged by lexicographic ordering of its constituent subexpressions from the bottom-most level, i.e., from the level of simple primaries assuming an ordering over the set of variables; among the function terminals, abs $\prec \div \prec$ mod $\prec$ uninterpreted function constants. (The symbol $\prec$ stands for the ordering relation "precedes".) As such, all function primaries, including those involving the uninterpreted ones, are ordered in a term in an ascending order of their arities.

(d) If the relational operator is $<$ or $<=$, then it will be converted to $>$ and $>=$, respectively, eg., `x` $<$ `20` will converted to `20 - x` $>$ `0`.

As a stronger condition should always precede a weaker condition in a construct like `if-else if-else`, violation of such precedence is detected using the rules given in Table 3.1. Let us consider the following example:

```
if (x > 15  ||  y < 10)
  sum = sum + 1;
else if (x > 15)
  sum = sum + 2;
else if (x > 20)
  sum = sum + 3;
else if (x > 45  &&  y < 15)
  sum = sum + 4;
else
  sum = sum + x;
```

Our automated precedence checker will compare the normalized conditions and detect the following violations in the code mentioned above:
(i) condition $\texttt{x} > \texttt{20}$ should precede condition $\texttt{x} > \texttt{15}$ (vide Rule 1),
(ii) condition $\texttt{x} > \texttt{15}$ should precede condition $\texttt{x} > \texttt{15} \;||\; \texttt{y} < \texttt{10}$ (vide Rule 2).
(iii) condition $\texttt{x} > \texttt{45 \&\& y} < \texttt{15}$ should precede condition $\texttt{x} > \texttt{15}$ (vide Rule 3), and
(iv) condition $\texttt{x} > \texttt{45 \&\& y} < \texttt{15}$ should precede condition $\texttt{x} > \texttt{20}$ (vide Rule 3).

We output such violations, if any, to the user to aid him/her in error correction.

Data structure for normalization

Actually, the method of equivalence checking of two given FSMDs involves some code transformation in the form of normalization of the computations as given in the paper by Sarkar and De Sarkar [75], in order to achieve some consistency in their representation.

The above paper suggests a data structure for normalized form (a normalized cell), called NCell for storing the normalized expressions. The equivalence checker works on the expressions stored in NCell.

```
struct normalized_cell
{
   NC *list;
   char type;
   int inc;
   NC *link;
};
```

Representation of the normalized expressions :

All normalized expressions are represented by tree structure which is implemented by linked lists [75]. Each node in the tree is a normalized cell consisting of the following four fields :

1. A LIST-pointer, which points to the entries at the same level of the tree or equivalently, at the same hierarchal level of an expression.

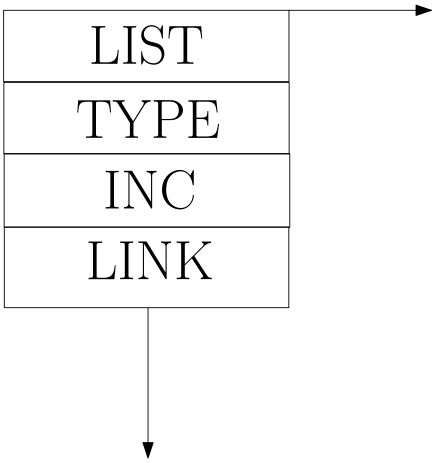


Figure 3.1: Depicting a normalized cell

2. A TYPE-field, which indicates the type of the cell. Some typical examples of the types are ‘S’ for normalized sum, ‘T’ for normalized term, ‘R’ for relational literal, etc. TYPE = ‘v’ indicates a program variable or more generally a symbolic constant.

3. An integer field INC, the meaning of which varies from type to type. For example, TYPE = ‘S’, INC = 4 means that the integer constant in the normalized sum is 4.

4. A LINK-pointer, which points to the leftmost successor of the node in question in the next level of the tree or equivalently, in the next syntactic level of the expression.

The difference between the LIST-pointer and the LINK-pointer is noteworthy. For example, the non-constant terms of a sum are connected by LINK-ing the first term to a normalized cell of TYPE ‘S’ and LIST-ing the other terms starting from the first term onwards.

Normalized sum: A normalized sum is a sum of terms with at least one constant term. Each term is a product of primaries with a non-zero constant primary. Each primary is a storage variable, an input variable.

Example of a normalized sum:

3 + 2 * a + 5 * x * y.

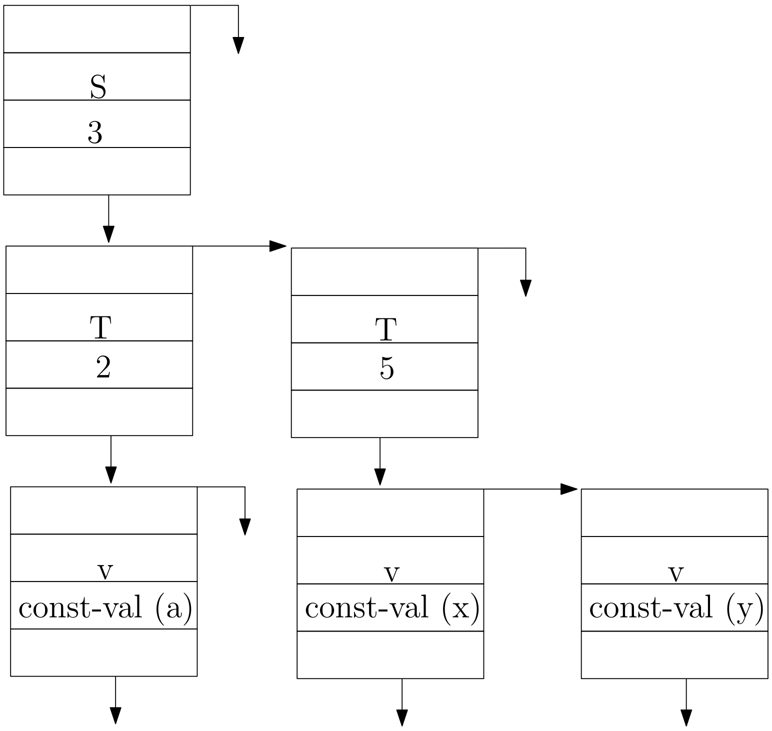


Figure 3.2: Depicting the expression 3 + 2 * a + 5 * x * y with list of normalized cells

3.2.3 Complexity analysis

In this subsection, we formally compute the worst case time complexity of finding the violation of precedence of conditions in `else-if` constructs as computed by our procedure mentioned above. The time complexity of our procedure is in the order of product of the following two terms: (i) the complexity of comparing two normalized (conditional) expressions, and (ii) the number of times such comparisons has to be done. If $||F||$ be the length of the normalized formula (in terms of the number of variables along with that of the operations in F), then the complexity of normalization of F is $O(2^{\|F\|})$ due to multiplication of normalized sums [12]. Since comparison of two normalized expressions involves a single traversal of the data structures of these two normalized expressions, the complexity of comparing two normalized expressions is also of order $O(2^{\|F\|})$. Let the number of conditions in an `else-if` construct be m; therefore, the number of comparisons that ought to be carried out is ${}^{m}C_2$ (i.e., the number of ways 2 objects can be chosen from a group of m distinct objects), which is $O(m^2)$. Thus, the overall time complexity of our procedure is $O(m^2 \cdot 2^{\|F\|})$.

Theorem 3 (Identification of redundant code)**.** *Let there be a nested if-else-if statement of the form*

```
if (C_1) {B_1}
else if (C_2) {B_2}
...
else if (C_i) {B_i}
...
else if (C_j-1) {B_j-1}
else if (C_j) {B_j}
else if (C_j+1) {B_j+1}
...
else {B_k}
```

for which the procedure reports that B_j is redundant; then, the following two if else-if statements are indeed equivalent

```
if (C_1) {B_1}
else if (C_2) {B_2}
...
```

```
else if (C_i) {B_i}
...
else if (C_j-1) {B_j-1}
else if (C_j) {B_j}
else if (C_j+1) {B_j+1}
...
else {B_k}
```

and

```
if (C_1) {B_1}
else if (C_2) {B_2}
...
else if (C_i) {B_i}
...
else if (C_j-1) {B_j-1}
else if (C_j+1) {B_j+1}
...
else {B_k}
```

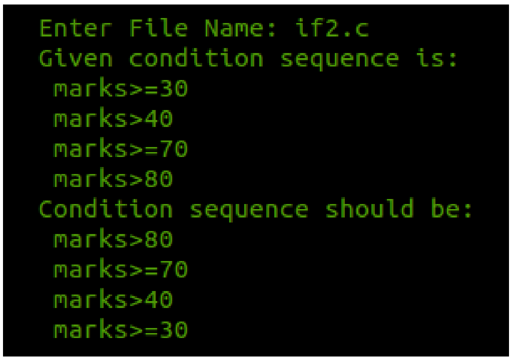
(i.e., the checking of the condition C_j and the block B_j can be dropped).

Proof: The procedure must have detected in step 2 some $i < j$, such that $C_j \Rightarrow C_i$. By contra position $\sim C_i \Rightarrow \sim C_j$.

The else-if $C_j\{B_j\}$ occurs in the $C_j = F$ exit and since $C_i = F \Rightarrow C_j = F$, the block B_j, which is executed for $C_j = T$ will never execute. ■

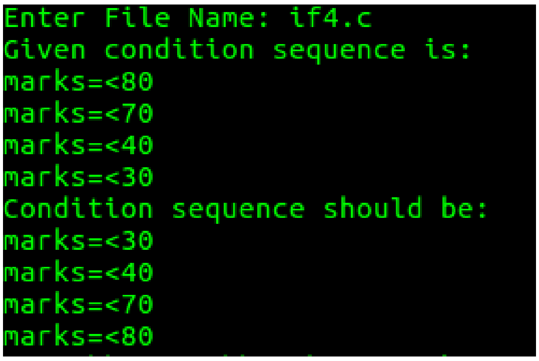
3.2.4 Results

The screen shots of results for various programs are shown below. Some of the results for the programs under the case 1 is shown in the figures 3.3 and 3.4. Some of the results for the programs under the case 2 is shown in the figures 3.5 and 3.6. Some of the results for the programs under the case 3 is shown in the figures 3.7 and 3.8. Some of the results for the programs under the case 4 is shown in the figures 3.9 and 3.10. Some of the results for the programs under the case 5 is shown in the figures 3.11 and 3.12. Some of the results for the programs under the case 6 is shown in the figures 3.13 and 3.14.



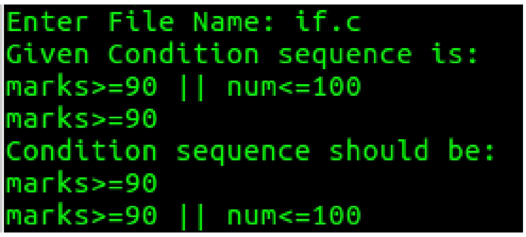
```
Enter File Name: if2.c
Given condition sequence is:
 marks>=30
 marks>40
 marks>=70
 marks>80
Condition sequence should be:
 marks>80
 marks>=70
 marks>40
 marks>=30
```

Figure 3.3: An example of case 1



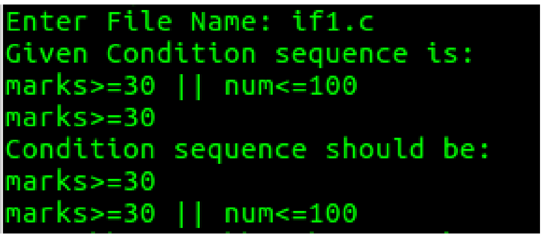
```
Enter File Name: if4.c
Given condition sequence is:
marks=<80
marks=<70
marks=<40
marks=<30
Condition sequence should be:
marks=<30
marks=<40
marks=<70
marks=<80
```

Figure 3.4: Another example of case 1



```
Enter File Name: if.c
Given Condition sequence is:
marks>=90 || num<=100
marks>=90
Condition sequence should be:
marks>=90
marks>=90 || num<=100
```

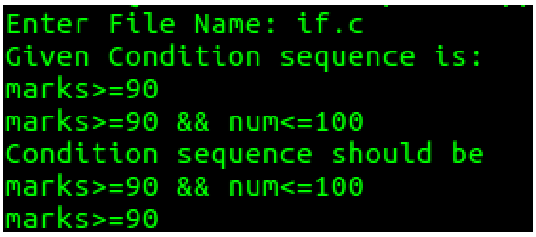
Figure 3.5: An example of case 2



```
Enter File Name: if1.c
Given Condition sequence is:
marks>=30 || num<=100
marks>=30
Condition sequence should be:
marks>=30
marks>=30 || num<=100
```

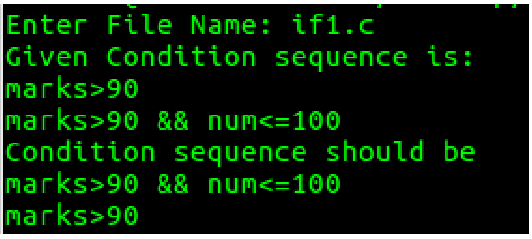
Figure 3.6: Another example of case 2

The table 3.3 summarizes the results of detecting incorrect order of sequences of the conditional construct. 10 sample programs for each of the cases were tested. Each of the sample program was chosen to have wrong sequencing of the conditional



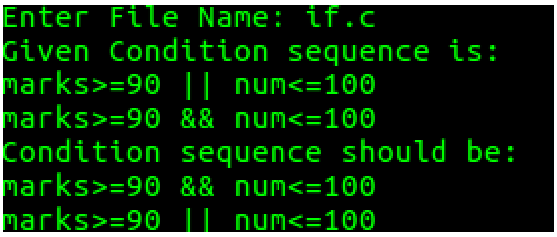
```
Enter File Name: if.c
Given Condition sequence is:
marks>=90
marks>=90 && num<=100
Condition sequence should be
marks>=90 && num<=100
marks>=90
```

Figure 3.7: An example of case 3



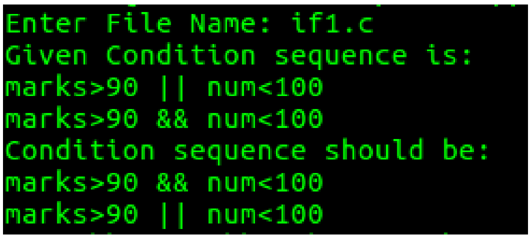
```
Enter File Name: if1.c
Given Condition sequence is:
marks>90
marks>90 && num<=100
Condition sequence should be
marks>90 && num<=100
marks>90
```

Figure 3.8: Another example of case 3



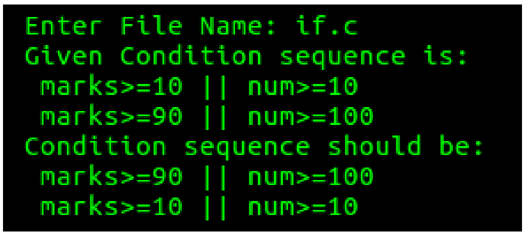
```
Enter File Name: if.c
Given Condition sequence is:
marks>=90 || num<=100
marks>=90 && num<=100
Condition sequence should be:
marks>=90 && num<=100
marks>=90 || num<=100
```

Figure 3.9: An example of case 4



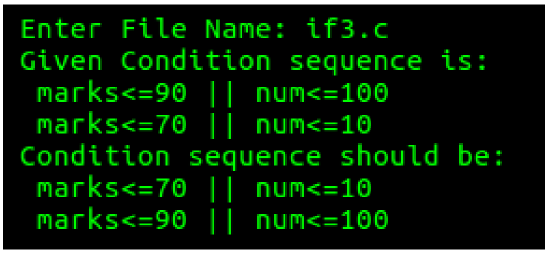
```
Enter File Name: if1.c
Given Condition sequence is:
marks>90 || num<100
marks>90 && num<100
Condition sequence should be:
marks>90 && num<100
marks>90 || num<100
```

Figure 3.10: Another example of case 4



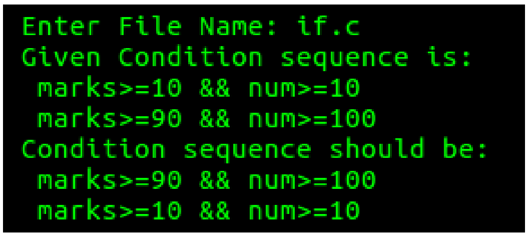
```
Enter File Name: if.c
Given Condition sequence is:
 marks>=10 || num>=10
 marks>=90 || num>=100
Condition sequence should be:
 marks>=90 || num>=100
 marks>=10 || num>=10
```

Figure 3.11: An example of case 5



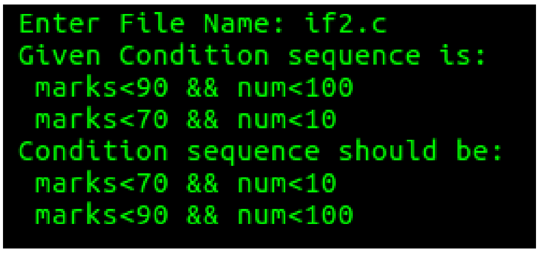
```
Enter File Name: if3.c
Given Condition sequence is:
 marks<=90 || num<=100
 marks<=70 || num<=10
Condition sequence should be:
 marks<=70 || num<=10
 marks<=90 || num<=100
```

Figure 3.12: Another example of case 5



```
Enter File Name: if.c
Given Condition sequence is:
 marks>=10 && num>=10
 marks>=90 && num>=100
Condition sequence should be:
 marks>=90 && num>=100
 marks>=10 && num>=10
```

Figure 3.13: An example of case 6



```
Enter File Name: if2.c
Given Condition sequence is:
 marks<90 && num<100
 marks<70 && num<10
Condition sequence should be:
 marks<70 && num<10
 marks<90 && num<100
```

Figure 3.14: Another example of case 6

construct. Our program was able to diagnose wrong ordering in test samples and suggest the correct ordering.

Table 3.3: Results for `else-if` constructs

Case	Precondition	Correct sequence	Incorrect sequence	# tested	# correct

1	$c_1 \Rightarrow c_2$	if (c_1) {...}
		else if (c_2) {...}
			if (c_2) {...}
			else if (c_1) {...}
				10	10

2	$c_3 \Rightarrow c_1$ or $c_3 \Rightarrow c_2$	if (c_3) {...}
		else if (c_1 || c_2) {...}
			if (c_1 || c_2) {...}
			else if (c_3) {...}
				10	10

3	$c_1 \Rightarrow c_3$ or $c_2 \Rightarrow c_3$	if (c_1&&c_2) {...}
		else if (c_3) {...}
			if (c_3) {...}
			else if (c_1&&c_2) {...}
				10	10

4	$c_1 \Rightarrow c_3$ and $c_2 \Rightarrow c_4$	if (c_1&&c_2) {...}
		else if (c_3 || c_4) {...}
			if (c_3 || c_4) {...}
			else if (c_1&&c_2){...}
				10	10

5	$c_1 \Rightarrow c_3$ and $c_2 \Rightarrow c_4$	if (c_1 || c_2) {...}
		else if (c_3 || c_4) {...}
			if (c_3 || c_4) {...}
			else if (c_1 || c_2) {...}
				10	10

6	$c_1 \Rightarrow c_3$ and $c_2 \Rightarrow c_4$	if (c_1&&c_2) {...}
		else if (c_3&&c_4) {...}
			if (c_3&&c_4) {...}
			else if (c_1&&c_2) {...}
				10	10

3.3 Variable mapping

Our objective is to find a mapping between variables in the student's program and the golden program, so that the variables of the student's program can be given the same name as those in the golden program. It is to be noted that equivalence checker works if the names of variables are same in the two FSMDs. We present below some examples to illustrate the variable mapping. Later an algorithm for variable mapping is provided.

3.3.1 Illustrative examples

Mapping dissimilar variables in two programs is explained with some examples given below.

Example 3.1. In this example we take two programs, in which the variables are assigned values in different steps. The two programs are as follows:

Listing 3.1: code 1

```
#include<stdio.h>
void main() {
   int a = 101, b = 2;
   if (a > 100) {
        a = a - 10;
   }
   if (b < 80) {
        b = b + 10;
   }
}
```

Listing 3.2: code 2

```
#include<stdio.h>
void main() {
   int x = 101, y;
   if (x > 100) {
      x = x - 10;
   }
   y = 2;
   if (y < 80) {
      y = y + 10;
   }
}
```

The FSMDs are given in the figures 3.15 and 3.16.

The steps of mapping variable names of the two FSMDs in the figures 3.15 and 3.16 are as follows.

1. In both the FSMDs, paths from the start state to the next cut-point are matched. Both paths involve an assignment operation but assign variables a, b in the first and x in the second FSMD. Hence, variable a is mapped with variable x, as both of them get same values.

2. Next, the condition $(a > 100)$ in the first FSMD is tried to match with the condition $(x > 100)$ in the second FSMD. This condition matches, so the variable val-

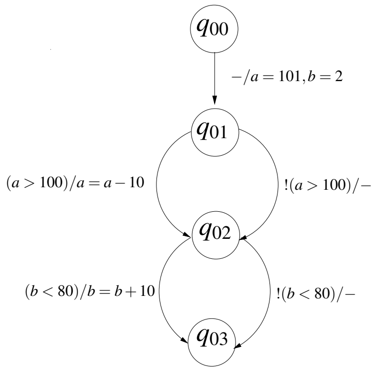


Figure 3.15: FSMD 1 of example 3.1

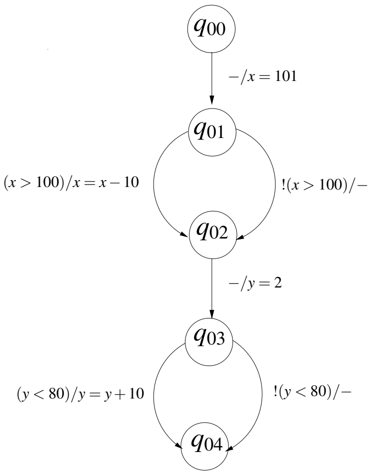


Figure 3.16: FSMD 2 of example 3.1

ues at the final state of the paths from the state where conditions were checked, are compared. The values at the respective final states of the paths are $a = 91$ and $b = 2$ in the first FSMD and $x = 91$ and $y = 2$ in the second FSMD. As a is already mapped with x, so nothing is to be done for them. As the values of b and y match, so they are mapped.

Mapping obtained as a result of above steps is as follows:

i) $a \mapsto x$, where the symbol $a \mapsto x$ is read as "a maps to x".

ii) $b \mapsto y$,

Example 3.2. In this example we take two programs which have different conditions. The two programs are as follows:

Listing 3.3: code 1

```
#include<stdio.h>
void main() {
    int a, b, c;
    printf("Enter a=");
    scanf("%d", &a);
    if (a == 0) {
        scanf("%d", &b);
        b++;
    }
    else {
        scanf("%d", &c);
        c--;
    }
}
```

Listing 3.4: code 2

```
#include<stdio.h>
void main() {
    int x, y, z;
    printf("Enter z=");
    scanf("%d", &z);
    z++;
    if (z != 0) {
        scanf("%d", &x);
        x--;
    }
    else {
       scanf("%d", &y);
       y++;
    }
}
```

The FSMDs are given in the figures 3.17 and 3.18.

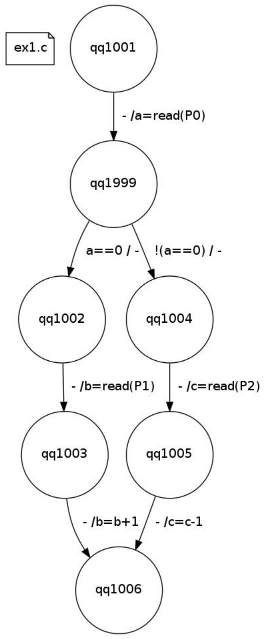


Figure 3.17: FSMD 1 of example 3.2

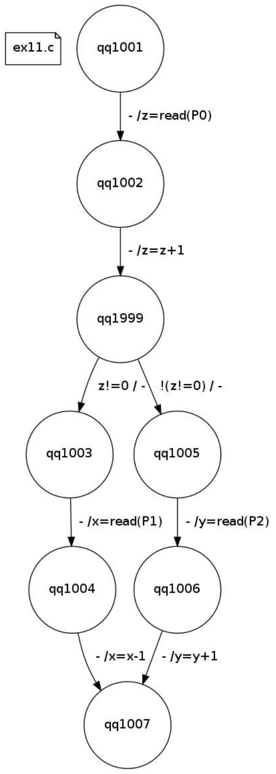


Figure 3.18: FSMD 2 of example 3.2

Steps of mapping variable names of the two FSMDs in the figures 3.17 and 3.18 are as follows.

1. In both the FSMDs, paths from the start state to the next cut-point are matched. Both involve a read operation to read variables a and z, respectively in the two FSMDs. Hence, variable a is mapped with variable z.

2. Next, the condition $(a == 0)$ is tried to match with the condition $(z! = 0)$ in the second FSMD. This condition does not match, so the next condition, $!(z! = 0)$, is tried. This condition matches with $(a == 0)$. Further, along the respective paths in the two FSMDs, variables y and b are read and incremented, so y is matched with b.

3. The condition $!(a == 0)$ of first FSMD is compared with $(z! = 0)$ in the second FSMD and conditions are found to match. Here c will be mapped with x as along the respective paths in the two FSMDs, variables y and b are read and decremented.

Mapping obtained as a result of above steps is as follows:

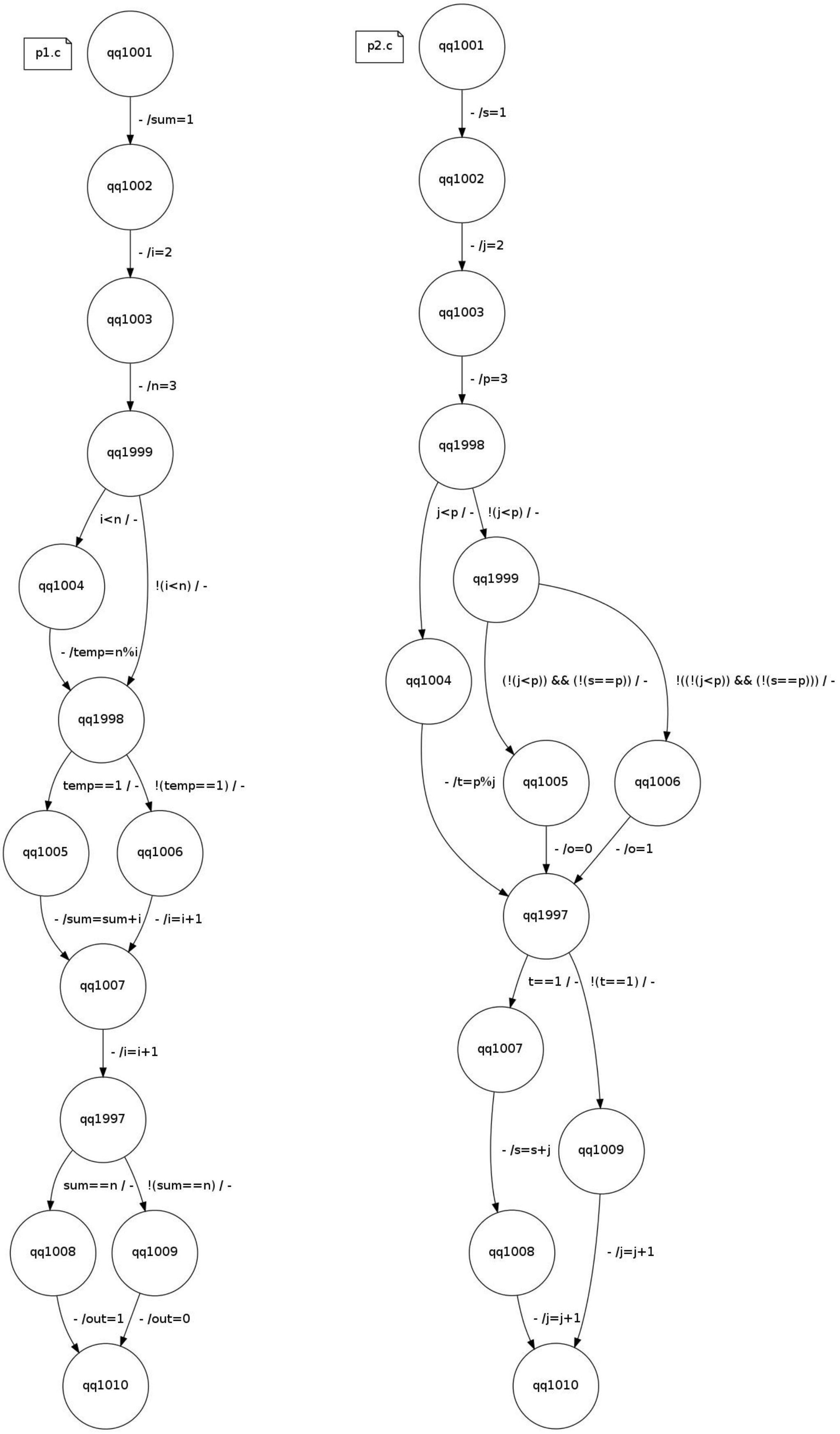
i) $a \mapsto z$,

ii) $b \mapsto y$,

iii) $c \mapsto x$.

Before we begin the next example we introduce the terms update vector and propagate vector of a path. Update vector of a path is the set of expressions, modifying the values of variables in a path, whereas, propagate vector is the set of conditions of execution and values of variables which are forwarded to the next path from the previous path. The terms are also explained later in the section 4.2.3 “Preliminary concepts of value propagation”

Example 3.3. In this example, we have taken FSMDs of two C programs as in figure 3.19. The first FSMD uses the variables $sum, i, temp, out$ and n and the second one uses the variables $s, j, temp1, ou$ and $n1$. Given these two FSMDs we have to find mapping of variables in them. The steps for mapping the variable names are as follows:



(a) (b)

Figure 3.19: Two FSMDs for example 3.3

1. For initial path from the start state to the next cut-point state in the two FSMDs, the path conditions α and β are both null. So conditions are matched and update vectors are $< sum = 1, i = 2, n = 3 >$ and $< s = 1, j = 2, n1 = 3 >$ with $\overline{V_0} = (-, < sum, i, n >)$ and $\overline{V_1} = (-, < s, j, n1 >)$ as the propagated vectors. Update values for *sum* and *s* are 1 and neither *sum* nor *s* is mapped, so *sum* will be mapped with *s* and similarly $j, n1$ will be mapped with i, n respectively.

2. For path condition $\alpha : (sum == n)$ which does not match with any condition β and as the propagated vectors $\overline{V_0} = (-, -)$ and $\overline{V_1} = (-, -)$ are null, mapping is not possible.

3. For path condition $\alpha : (!(sum == n))$ which partially matches with $\beta : (!(j < n1)\&\&!(s == n1))$ and $\overline{V_0} = (!(sum == n), < out >)$ and $\overline{V_1} = (!(j < n1)\&\& !(s == n1), < ou >)$. The update values are $out = 0$ and $ou = 0$. Both have the same update values and both are not mapped previously, so *out* is mapped with *ou*.

4. For path conditions $\alpha : (temp == 1)$ and $\beta : (temp1 == 1)$, no mapping is done as $temp1$ is not yet defined.

5. For path conditions $\alpha : (!(temp == 1))$ and $\beta : (!(temp1 == 1))$, no mapping is done as $temp1$ is not yet defined.

6. For path condition $\alpha : (i < n)$ which matches with condition $\beta : (j < n1)$ and propagated vectors are $\overline{V_0} = ((i < n), < temp >)$ and $\overline{V_1} = ((j < n1), < temp1 >)$, The variable values are $temp = n\%i$ and $temp1 = n1\%j$, thus the update vectors are same, $temp1$ is now defined and hence $temp1$ is mapped with $temp$.

7. For path condition $\alpha : (!(i < n))$ which matches with condition $\beta : (!(j < n1))$, as the propagated vectors $\overline{V_0} = (-, < - >)$ and $\overline{V_1} = (-, < - >)$ are null, so no mapping is done.

Mapping obtained as a result of above steps is as follows:

i) $s \mapsto sum$,

ii) $i \mapsto j$,

iii) $n \mapsto n1$,

iv) $out \mapsto ou$, and

v) $temp \mapsto temp1$.

3.3.2 Variable mapping algorithm

Given two FSMDs say $M_s = \langle Q_s, q_{s,0}, I_s, V_s, O_s, \tau_s : Q_s \times 2^{S_s} \rightarrow Q_s, h_s : Q_s \times 2^{S_s} \rightarrow U_s \rangle$, and $M_g = \langle Q_g, q_{g,0}, I_g, V_g, O_g, \tau_g : Q_g \times 2^{S_g} \rightarrow Q_g, h_g : Q_g \times 2^{S_g} \rightarrow U \rangle$, variable mapping tries to obtain a mapping between the variables of M_s and M_g. The variable map is thus a mapping between the variables of the sets I_s and I_g, V_s and V_g and O_s and O_g. Algorithm 7 obtains the variable map $\mathcal{M}$: $V_s \rightarrow V_g$=$\{\langle v_s, v_g \rangle | v_s \in (V_s \cup I_s \cup O_s)$, $v_g \in (V_g \cup I_g \cup O_g)\}$.

The step 4(a) is elaborated as follows.

[Step 4(a)] If C_s and C_g, the conditions of path p_s and p_g respectively, are equal after applying the map established so far, then add all the transitions of p_s and p_g in AT_s and AT_g respectively and then find mapping between the two lhs variables of statements (aka “actions” here) from AT_s and AT_g, according to the function Map_variable(), where AT_s and AT_g stand for action tables for student and golden programs respectively and they are used to store actions from the paths p_s and p_g respectively of the two programs.

Then every statement in AT_g is compared with statements from AT_s, beginning with

Algorithm 7: Variable mapping

```
    Input: $M_s$, FSMD of student's program and $M_g$, FSMD of teacher's program;
    Output: Map $\mathcal{M}$=$\{\langle v_s, v_g\rangle | v_s \in (V_s \cup I_s \cup O_s), v_g \in (V_g \cup I_g \cup O_g)\}$
L1  [Step 1] Initialize $V_s$, $V_g$ for storing statements of $M_s$, $M_g$.
    /*Step 2 is implemented by the function Find_path(), which is
      a depth first search based function described later.
      Find_path() uses Divide_path() and Store_path(), which are
      described later                                              */
L2  [Step 2] Insert cut-points in $M_s$ and $M_g$ and construct the respective path covers,
    $P_s$ for $M_s$ and $P_g$ for $M_g$ using function Find_path().
L3  [Step 3] $\mathcal{M} \leftarrow \phi$ (Null set)
    /*Step 4 is implemented by the function Propagate(), which
      uses the functions Check(), Check_condition(),
      Map_variable(), Map_assign_variable(),
      Relation_equivalence() to implement the steps 4(a) & (b).
      All these functions are described later                      */
L4  [Step 4] Select a path $p_s$ and $p_g$ from $P_s$ and $P_g$ respectively.
L5  [Step 4(a)] If $C_s$ and $C_g$, the conditions of path $p_s$ and $p_g$ respectively, are equal
    after applying the map $\mathcal{M}$ established so far, then add all the transitions in $V_s$
    and $V_g$ and find mapping between them according to the function Map_variable
    and Map_assign_variable (see the respective algorithms).
L6  $\mathcal{M} \leftarrow \mathcal{M} \cup \{v_s, v_g\}$
L7  [Step 4(b)] Else select next $C_{s_i}$ such that $C_g$ and $C_{s_i}$ are the same and go to L5,
    step 4(a).
L8  [Step 5] Output $\mathcal{M}$.
```

the statement at the start of AT_s, till a match was found.

For this, the function Map_variable will take two statements called $action_1$ and $action_2$ from AT_s and AT_g respectively as input. It will map the variables of $action_1$ and $action_2$ as per the following cases.

Case 1: If rhs of both $action_1$ and $action_2$ are read statements and variables of $action_1$ or $action_2$ are not mapped , then map the rhs variables.

Case 2: Else, if the variables at the lhs of $action_1$ or lhs of $action_2$ are not mapped, then map the variable in the lhs of $action_1$ to the variable in the lhs of $action_2$, only if rhs expressions are equivalent after applying the map established so far.

/* Mapping the already mapped variable */

Case 3: The function Map_assign_variable will take a statement from AT_s and map the lhs variable of the statement with a variable 'v' of the golden program, in case the variable at rhs is already mapped with variable 'v'.

$\mathcal{M} \leftarrow \mathcal{M} \cup \{v_s, v_g\}$

An inherent limitation of variable mapping is that it is not final and correct in all cases, but it is still to be done as variable mapping enables us to start equivalence checking, which would not be possible if the two programs have different variable names.

Theorem 4 (Support for variable mapping)**.** *If $\langle v_{s,i}, v_{g,j} \rangle \in \mathcal{M}$, then there exists at least one pair of paths p_s of FSMD M_s and p_g of FSMD M_g such that the value of $v_{s,i}$ computed by p_s is equal to the value of $v_{g,j}$ computed by p_g.*

Proof: Let there exist a pair of $\mathcal{M}$, $\langle v_{s,i}, v_{g,j} \rangle \in \mathcal{M}$ for which there are no paths p_s and p_g, which compute their values equally. Since pairs are put in $\mathcal{M}$ in step 4(a) on finding at least one pair of paths which compute equal values for the variables, such pairs cannot occur in $\mathcal{M}$. ■

In the proof of theorem 4, the second sentence assumes that pairs will be put in $\mathcal{M}$. What is the guarantee that this will happen? There is no guarantee. $\mathcal{M}$ can be empty (which is not desirable, but may happen if one or both the programs do not use input or program variables). The second sentence asserts that pairs are put in $\mathcal{M}$ in step 4a. Note that step 4a is a conditional statement; if the condition does not hold for any path P_s and P_g, then $\mathcal{M}$ can be empty and hence, the theorem holds vacuously. The proof concerns itself with the non-vacuous cases.

Important modules in the program for variable mapping

The steps of the variable mapping algorithm given above have been implemented as various modules, a brief description of the important modules is given below.

Find paths in FSMD

The function Find_path will take an FSMD as an input and will find all the paths of FSMD and store them into PathList[]. It may be noted that a path is a sequence of states from one cut-point to the next, represented in the program by a structure Path. PathList[i] will contain the i-th Path. Steps of function Find_path are as follows.

Step 1: For all the states State in the FSMD, if State is a cut-point, then push it into Stack.

Step 2: For all the outgoing transitions from State do the following: push the state at the other end of the transition (called TransitionEndState) into Stack and call the function Divide_path.

Divide path

The function Divide_path will take an FSMD, State, PathList[] and Stack as an input, where State will act as the current state from which a path of FSMD will be generated

Algorithm 8: Find paths in FSMD

```
Function Find_path (FSMD, PathList[])
forall states State in FSMD do
    if State is a cut-point then
        push State into Stack;
        forall outgoing transitions from State do
            push TransitionEndState into Stack;
            Divide_Path(FSMD, TransitionEndState, PathList[], Stack);
return;
```

by Divide_path. Eventually, Stack will hold all the states of a path, where the bottom most entry of Stack will contain the starting state of the path and the top most entry of Stack will contain the final state of the path. Function Divide_path will generate all the possible paths from the current state. Steps of the function Divide_path are as follows:

Step 1: If State is a cut-point, then the function Store_path is called.

Step 2: Otherwise, for all the TransitionEndStates of State, the TransitionEndState is pushed into Stack and Divide_path is called recursively.

Store path

Store_path will pop all the states from Stack and put them in Path. Then Path will then be added to PathList[].

Algorithm 9: Divide path

```
Function Divide_path(FSMD, State, PathList[], Stack)
if State is a cut-point then
    Store_path(PathList[], Stack);
else
    forall outgoing transitions from State do
        push TransitionEndState into Stack;
        Divide_path(FSMD, TransitionEndState, PathList[], Stack);
return;
```

Algorithm 10: Store paths in path node

```
Function Store_path(PathList[], Stack)
forall states in Stack do
    store state in Path;
add Path to PathList[NumberOfPaths + 1] ;
NumberOfPaths = NumberOfPaths + 1;
return;
```

Map variables

First we describe here the mapping operation, which will be used in the subsequent cases:

Mapping operation : In this part, we consider that a path from golden program and a path from student program have been selected such that the condition at their starting states have matched. All the statements from golden program are placed in ActionTable_G[], all the statements from student program are stored in ActionTable_-S[]. Variable count1 is assigned the number equal to the number of statements appearing in golden program. Similarly, the variable count2 is assigned the number equal to the number of statements appearing in the student program. Then, every statement in the ActionTable_G[] is compared with the statements from the ActionTable_S[], beginning with the statement at the start of the ActionTable_S[], till a match was found. While comparing the statements, the following cases may be encountered:

Case 1 : If the function Check has returned true for the ActionTable_G[i] and ActionTable_S[j], then the function Map_variable is called to map the variables of the statements i, j of ActionTable_G[i], ActionTable_S[j].

Case 2 : If the RHS of ActionTable_S[j] is term (assignment statement) and have only one variable which is initialized with a constant value different from golden program and if the variable at RHS is already mapped and the variable on the LHS is not, then map the LHS variable using the function Map_assign_variable.

This completes our description of the mapping operation, which is referred in the description of function propagate below.

For the two inputs to function propagate, *i)* PathList_G[] of golden program and *ii)* PathList_S[] of student program, created by function Find_path, the function propagate will create an ActionTable. The ActionTable will contain ActionTable_G[] for the golden program and ActionTable_S[] for the student's program, for every path in both the PathLists . The ActionTable_G[] and ActionTable_S[] are used to hold all the statements appearing in the golden and student programs, respectively. Other inputs for function propagate are symbol tables from golden program (SymbolTable_G) and student program (SymbolTable_S). Symbol table will contain all the variable names and their information (variable name, data type, variable values etc.). Function propagate will output all possible variable mappings between both, the golden program and the student program. Steps of the function propagate are as follows:

Case 1 : The starting state of the first path of an FSMD does not have any condition, hence, we need not use the function Check_condition before mapping variable.

The variable mapping operation as described above, can be done in this case straight away.

Case 2 : In case the path under consideration is not starting from the starting state of FSMD, such a path must have originated from a cut point, due to some condition or its negation. Mapping of variables is done only if the paths of golden and the student program have originated under the same conditions. Hence, at first, the condition of the two paths are checked before proceeding with previously described mapping operation.

Check action type

This module will check whether two given actions are of same type or not. following are the cases:

- Types of actions are 'r', 'w', 'C', where 'r' stands for 'READ'(scanf), 'w' for

Algorithm 11: Map variables based on propagation vector

```
Function Propagate(PathList_G[], SymbolTable_G, PathList_S[], SymbolTable_S)
MappedVars=0;
forall $i = 0$ to NumberOfPaths in PathList_G[] do
    if i==0 then
        forall Transition actions in Path at PathList_G[i] do
            push transition action in ActionTable_G[];
        forall Transition actions in Path at PathList_S[i] do
            push transition action in ActionTable_S[];
        forall j=0 to all actions in ActionTable_G[] do
            forall k=0 to all actions in ActionTable_S[] do
                if Check( ActionTable_G[j], ActionTable_S[k]) return true then
                    Map_Variable( ActionTable_G[j], ActionTable_S[k]);
                    MappedVars++;
            else if the RHS of ActionTable_S[j] is a term having only one variable which is initialized with a
              constant value then
                if the variable at RHS is already mapped and the LHS variable is not then
                    Map_assign_variable(ActionTable_S[k]);
                    MappedVars++;
    else
        C1 = condition at the starting state of Path at PathList_G[i];
        forall Transition actions in Path at PathList_G[i] do
            push transition action in ActionTable_G[];
        forall  j=1 to all paths in PathList_S[] do
            C2 = condition at the starting state of Path at PathList_S[j];
            if Check_Condition (C1, C2) return true then
                forall Transition actions of Path at PathList_S[j] do
                    push transition action in ActionTable_S[];
                forall l=0 to all actions in ActionTable_G[] do
                    forall m=0 to all actions in ActionTable_S[] do
                        if  Check( ActionTable_G[l], ActionTable_S[m]) return true then
                            Map_Variable( ActionTable_G[l], ActionTable_S[m]);
                            MappedVars++;
                        else if the RHS of ActionTable_S[j] is a term having only one variable which is
                          initialized with a constant value then
                            if the variable at RHS is already mapped and the variable on the LHS is not
                              then
                                Map_assign_variable(ActionTable_S[m]);
                                MappedVars++;
return;
```

‘WRITE’(printf) and ‘C’ for ‘CONSTANT’(initialization with a value). If both the actions are of same type, it will return true.

- If the left hand side and the right hand side of both the actions is the same, then also it will return true.

Algorithm 12: Check action type

```
Function Check(ActionTable_G[i], ActionTable_S[j])
if both ActionTable_G[i] and ActionTable_S[j] are of type ‘r’ then
    return true;
else if both ActionTable_G[i] and ActionTable_S[j] are of type ‘w’ then
    return true;
else if both ActionTable_G[i] and ActionTable_S[j] are of type ‘C’ then
    if initialized value of variable at ActionTable_G[i] is equal to initialized
      value of variable at ActionTable_S[j] then
        return true;
    else
        return false;
else if left hand side of ActionTable_G[i] and ActionTable_S[j] is equal then
    if right hand side of ActionTable_G[i] and ActionTable_G[j] is equal then
        return true;
    else
        return false;
else
    return false;
return;
```

It may be noted that the symbol tables S and G are generated as part of parser code for generating the FSMD. Symbol tables S and G will simply be used as reference to keep track of all the variables used in the program. S and G here stand for student's program and golden program respectively.

The word "action" here refers to a statement in the program.

The action types are used in algorithm 12 to find out whether the expressions in the programs being compared are same or not. Action types are 'r', 'w', 'C', where 'r' stands for 'READ'(scanf), 'w' for 'WRITE' (printf) and 'C' for 'CONSTANT' (initialization with a value).

Check condition

This function is used to check whether two conditions are equivalent or not. Function Check_condition will take two conditions (C1 and C2) as input. The conditions may be compound. Conditions and expressions are stored in normalized form to ensure that the same condition written in different forms, will eventually be converted to a unique form. For programming convenience, different operators are given integer values. 74, 76, 77 are for NOT, AND, OR respectively, where OR and AND are logical operators. There are several cases, where two conditions will be equal as follows:

1. If the normalized form of both the conditions are equivalent and all the constant values used in expression are also equal.
2. If one of C1 or C2 is of type 76 or 77, it means both the conditions are not perfectly matched, but partial condition matching is possible.
3. If both the conditions are of opposite types, then there may arise several cases:
    (a) If the normalized value of type of C1 is 74, then there are relational operators in conditions, hence, we check for relational operator equivalence with function Relation_equivalence.

   i. If the function Relation_equivalence returns true, then if type of C2 is 74, it shows that the condition is composite. Hence, compare the tree of normalized cells (NC) structure of both the conditions and return result accordingly.

   ii. If type of C2 is 76 or 77, then sub parts of C2 will be checked with C1, using function Check_condition.

(b) If the normalized value of type of C2 is 74, then there are relational operators in conditions, hence, we check for relational operator equivalence with function Relation_equivalence.

   i. If the function Relation_equivalence returns true, then if type of C1 is 74, it shows that the condition is composite. Hence, compare the tree of normalized cell (NC) structure of both the conditions and return result accordingly.

   ii. If type of C1 is 76 or 77 then, sub parts of C1 will be checked with C2, using function Check_condition.

Explanation for line 25 algorithm 13 is as follows. A condition will be divided into sub-parts if it is composite condition having multiple && or || conjuncts.

Mapping the variables in the statements

Function Map_variable will take two actions (action1 and action2) from ActionTable_G[] and ActionTable_S[] respectively as an input. It will map variables of action1 and action2. Cases where mapping between variables of action1 and action2 is possible are as follows:

Case 1: If RHS of both action1 and action2 are read statements and variables of action1 or action2 are not mapped, then map variable.

Algorithm 13: Check conditions

```
Function Check_condition (Condition1 C1, Condition2 C2)
if types of both C1 and C2 are same and value of normalized expression type in both conditions are same then
    if value of normalized actions of both conditions are equal then
        return true;
else if value of normalized action in any of condition C1 or C2 is 76 or 77 then
    partial checking of Condition1 and Condition2;
    return true;
else if If either of C1 or C2 is of type ‘O’ then
    if value of normalized action in C1 is 74 then
        if Check_Relation_Equi (C1, C2) then
            if C2 type is 74 and both normalized conditions are equivalent then
                return true;
            else if C2 type is 76 or 77 then
                Check_condition(sub part of C1, sub part of C2);
            else
                if normalized values of both C1 and C2 are equivalent then
                    return true;
                else
                    return false;
    else if Value of normalized action in C2 is 74 then
        if Check_Relation_Equi(C1, C2) then
            if C1 type is 74 and both normalized conditions are equivalent then
                return true;
            else if C1 type is 76 or 77 then
                Check_condition(sub part of C1, sub part of C2);
            else
                if normalized values of both C1 and C2 are equivalent then
                    return true;
                else
                    return false;
else if Both conditions are NULL then
    return true;
else
    return false;
return;
```

Case 2: variable at the LHS of action1 or LHS of action2 is not mapped then map variable in the LHS of action1 to the variable in the LHS of action2.

Algorithm 14: Map variables

```
Function Map_variable (ActionTable_G[i], ActionTable_S[j])
if RHS of both ActionTable_G[i] and ActionTable_S[j] are read statements then
    if Variables of ActionTable_G[i] are not mapped or variables of
      ActionTable_S[j] are not mapped then
        map the variable read in ActionTable_G[i] to the variable read in
          ActionTable_S[j];
else if the variable in LHS of ActionTable_G[i] or LHS of ActionTable_S[j] is
  not mapped then
    map the variable in the LHS of ActionTable_G[i] to the variable in LHS of
      ActionTable_S[j];
return;
```

Mapping with the already mapped variable

Function Map_assign_variable will take a statement from ActionTable_S[] and map the variable of that statement with already mapped variable of ActionTable_G[]. It will search the variable at RHS of action of student program with already mapped variable list of golden program. If such a variable is found, map that variable at LHS of ActionTable_S[i] to variable which is found mapped with golden program.

Check equivalent relational operators

Function Check_Relation_Equi will take two relational operators as an input. If both the operators are same, it will return true.

Algorithm 15: Map variable of assignment statement

```
Function Map_assign_variable(ActionTable_S[i])
if variable at RHS of ActionTable_S[i] is already mapped with a variable ‘v’ of
  golden program then
    map the variable at LHS of ActionTable_S[i] to ‘v’.
```

Algorithm 16: Relation equivalence

```
Function Check_Relation_Equi(Relational operator C1, Relational operator
  C2)
if Relational operator C1 == C2 then
    return 1;
else
    return 0;
return;
```

3.3.3 Demonstration of the algorithm

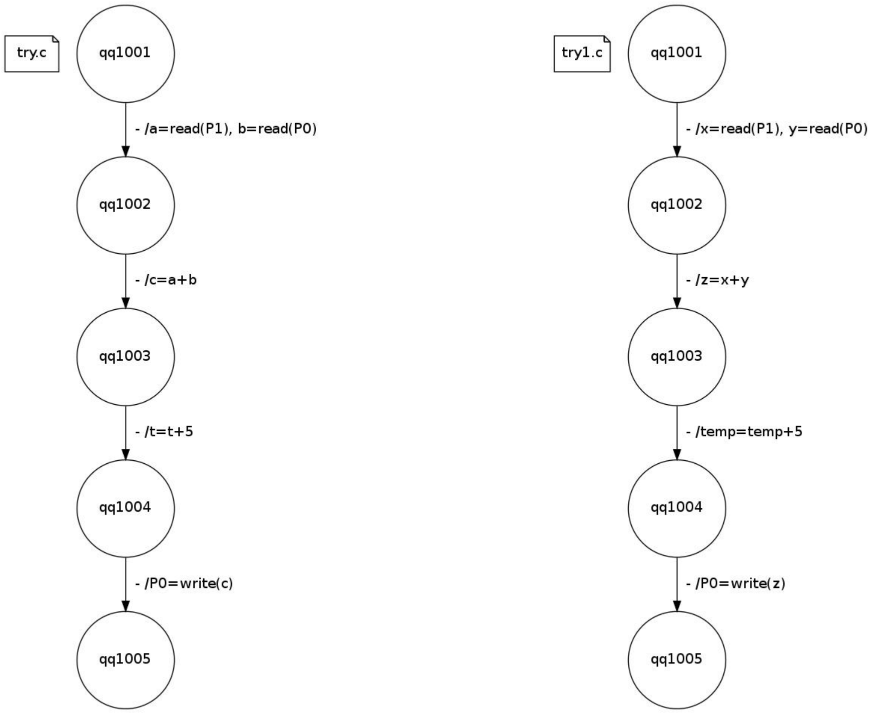
In this section we first demonstrate the working of variable mapping algorithm for the following programs. The FSMDs of the two programs are shown in the figure 3.20.

Listing 3.5: code 1

```
#include <stdio.h>
int main() {
 int a, b, c;
 int t;
 printf("Enter a and b: ");
 scanf("%d%d", &a, &b);
 c = a + b;
 t = t + 5;
 printf("Addition: %d", c);
 return 0;
}
```

Listing 3.6: code 2

```
#include <stdio.h>
int main() {
 int x, y, z;
 int temp;
 printf("Enter x and y: ");
 scanf("%d%d", &x, &y);
 z = x + y;
 temp = temp + 5;
 printf("Addition: %d", z);
 return 0;
}
```



(a)

(b)

Figure 3.20: Two FSMDs: demonstration of variable mapping program

We present below the output of the variable mapping program.

```
C1=NULL and C2=NULL
Trying to map P1=read(a)= P1=read(x)
Mapping var =2 to var=2
C1=NULL and C2=NULL
Trying to map P0=read(b)= P0=read(y)
Mapping var =3 to var=3
C1=NULL and C2=NULL
Trying to map c=a+b= z=x+y
Mapping var =4 to var=4
C1=NULL and C2=NULL
Trying to map t=t+5= temp=temp+5
C1=NULL and C2=NULL
Trying to map t=t+5= P0=write(z)
C1=NULL and C2=NULL
Trying to map P0=write(c)= temp=temp+5
C1=NULL and C2=NULL
Trying to map P0=write(c)= P0=write(z)
Mappings are as follows
a=x
b=y
c=z
```

Explanation of the execution: As a and x are input variables, read using scanf function, so they are mapped first. Similarly, the variables b and y are mapped for being the input variables, which get their values now. The variables c and z are mapped next as in c=a+b, both a and b have been assigned values, similarly in z=x+y, both x and y have been assigned values. Next it tries to map t and temp by comparing t=t+5 and temp=temp+5. Mapping is not possible as both t and temp are uninitialized. Algorithm further tries to map them by comparing them with the remaining statements, while traversing the FSMD. However they cannot be mapped as they remain uninitialized. The mapping done thus is as follows.

$\mathtt{a} \mapsto \mathtt{x}$

$\mathtt{b} \mapsto \mathtt{y}$ and

$\mathtt{c} \mapsto \mathtt{z}$

3.3.4 Various cases of variable mapping

As explained above, the mapping can be done for variables in two programs. Different cases can be possible for mapping as follows.

- When the condition of two programs is not the same.
- A program has more variables than the other program.
- Program may contain an uninitialized variable.
- Wrong variable display at the output.
- Wrong input data is used in one program.
- Missing loops.
- Independent variables can be mapped anywhere.

We discuss some of the cases with an example of each below.

I. When the condition of two programs is not the same

Two programs with different conditions are as follows:

Listing 3.7: code 1

```
 #include<stdio.h>
void main() {
   int a,b,c;
   printf("Enter a value");
   scanf("%d",&a);
   if(a == 0) {
        scanf("%d", &b);
        b++;
   }
   else {
        scanf("%d", &c);
        c--;
   }
}
```

Listing 3.8: code 2

```
#include<stdio.h>
void main() {
   int x,y,z;
   printf("Enter a value");
   scanf("%d", &z);
   z++;
   if(z != 0){
        scanf("%d", &x);
        x--;
   }
   else {
        scanf("%d", &y);
        y++;
   }
}
```

The FSMDs are given in the figure 3.21 given below.

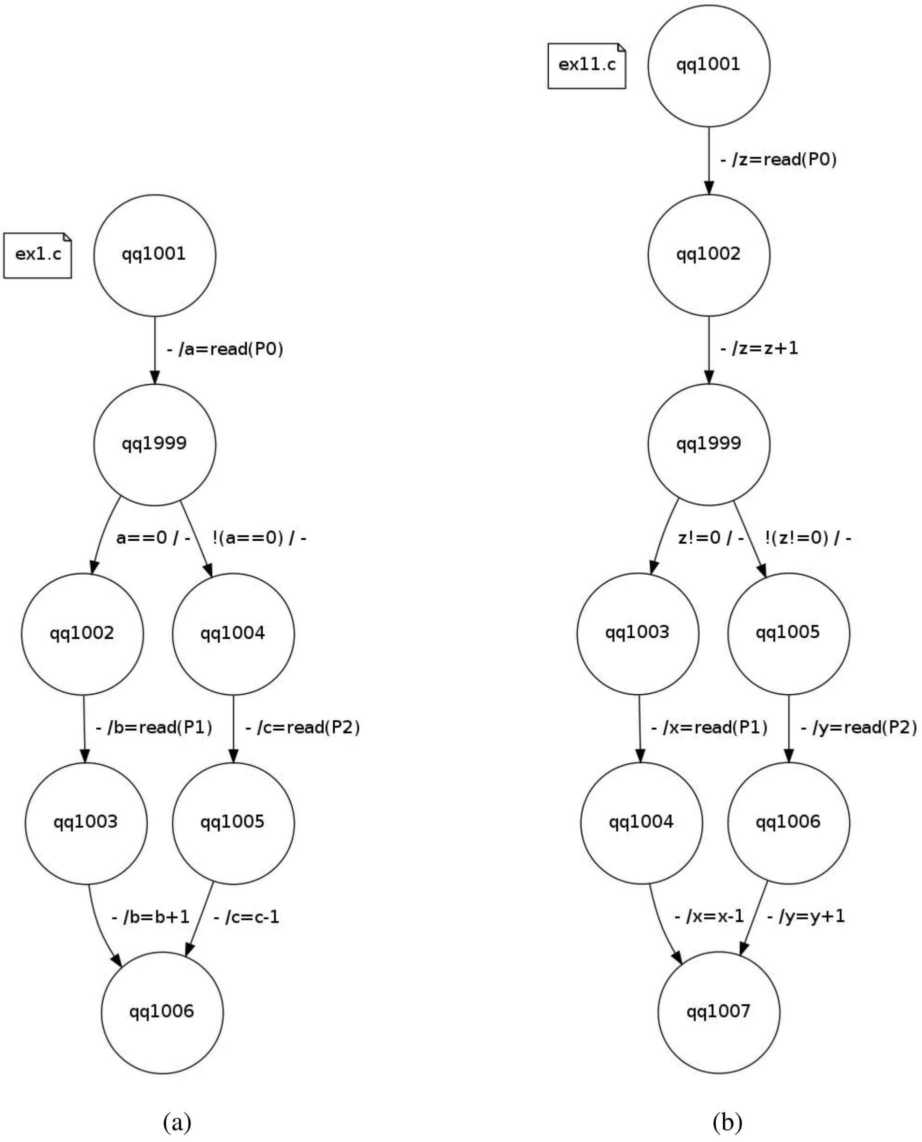


Figure 3.21: Two FSMDs: Case I, variable mapping

Evaluation steps

1. Path 0 of both path-covers are matched. Both are read operations, so we can say that variable *a* will map with variable *z*.

2. Path 1 of first path-cover tries match condition with path 1 of second path-cover but, condition is not matched. So, next path condition of second path-cover tries

to match. The condition is matched and y is matched with b.

3. Path 2 of first path-cover tries to match condition with path 1 and condition matched. So c will map with x.

Final mappings of the given programs are as follows: $a \mapsto z$
$b \mapsto y$
$c \mapsto x$

II. A program has more variables than the other program

Let the two programs be as follows:

```
#include <stdio.h>

/* Problem 3
 * Friendly Numbers are those which have same abundancy.
 * Abundancy of a number is the ratio of sum of its divisors and
 * the number itself.
 * For Ex : Abundancy of 6 is (1+2+3+6)/6=2
 * Abundancy of 28 is (1+2+4+7+14+28)/28=2
 * So, 6 and 28 is a friendly number pair.
 * Write a program takes two numbers as input and decides whether
 * they are friendly numbers or not.
 */

int main()
{
  int x, y, sumx, sumy,count0=0,count1=1;
  double abunx, abuny;
  int i;

  printf("Enter two numbers:");
  scanf("%d%d",&x,&y);

  sumx = 0;
  for(i = 1; i <= x; i++)
  {
    if((x%i) == 0)
      sumx = sumx + i;
  }

  sumy = 0;
  for(i = 1; i <= y; i++)
  {
    if((y%i) == 0)
      sumy = sumy + i;
  }

  abunx=(sumx)/x;
  abuny=(sumy)/y;

  if(abunx == abuny)
  {
    printf("%d",count1);
  }
  else
  {
    printf("%d",count0);
  }
  //printf("Abundunancy for x=%d and y=%d",abunx,abuny);
  return 0;
}


#include<stdio.h>
int main()
{
  int i,a,b,sa,sb,cou0=0,cou1=1;
  printf("Enter 2 nos\n");
  scanf("%d%d",&a,&b);
        sa=0;sb=0;
  for(i=1; i<=a; i++){
     if((a%i)==0){
            sa+=i;
     }
  }
  for(i=1; i<=b; i++){
     if((b%i)==0){
            sb+=i;
     }
  }
  sa=sa/a;
  sb=sb/b;
  if(sa==sb) {
      printf("%d",cou1);
  }
  else {
      printf("%d",cou0);
  }
  return 0;
}
```

The FSMDs are given in the following figures, 3.22 and 3.23.

Evaluation steps

1. Path 0 of first path-cover it finds *count*0 and *cou*0 are initialized, so they are mapped.
2. Path 0 of first path-cover it finds *count*1 and *cou*1 are initialized, so they are mapped.
3. Path 0 of first path-cover it finds *x* and *a* are read statements, so they are mapped.
4. Path 0 of first path-cover it finds *y* and *b* are read statements, so they are mapped.
5. Path 0 of first path-cover it finds *sumx* and *sa* are initialized, so they are mapped.
6. Path 0 of first path-cover it finds *i* and *i* are initialized, so they are mapped.
7. Path 0 of first path-cover it finds *sumy* and *i* are initialized, so they are mapped.

So, the mappings are as follows:

$count0 \mapsto cou0$

$count1 \mapsto cou1$

$x \mapsto a$

$y \mapsto b$

$sumx \mapsto sa$

$i \mapsto i$

$sumy \mapsto i$

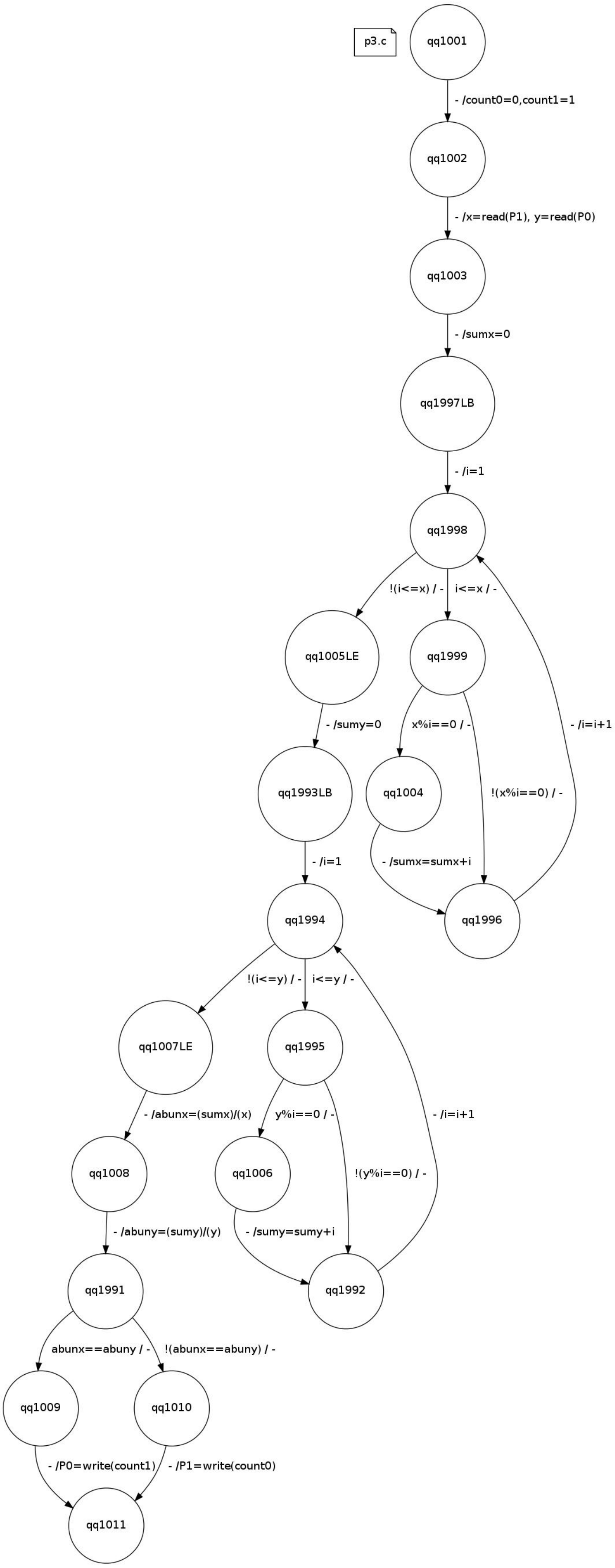


Figure 3.22: FSMD 1: Case II, variable mapping

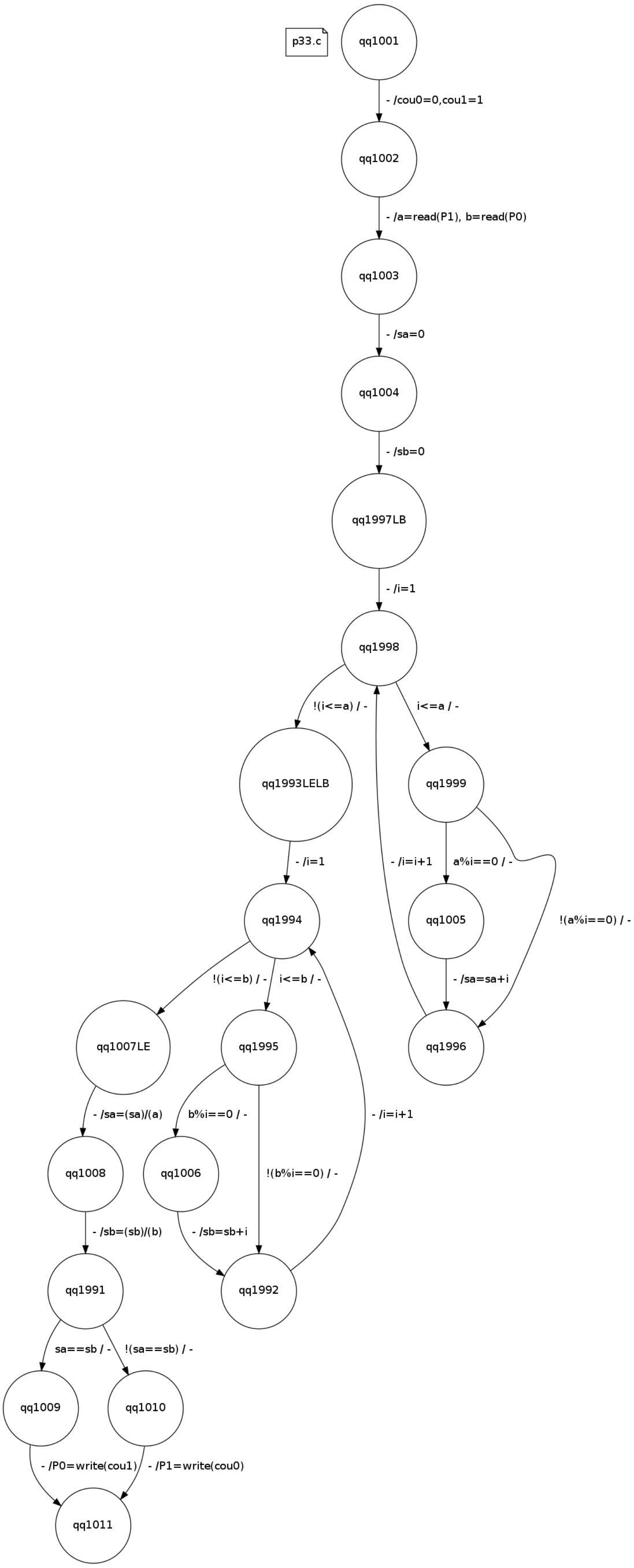


Figure 3.23: FSMD 2: Case II, variable mapping

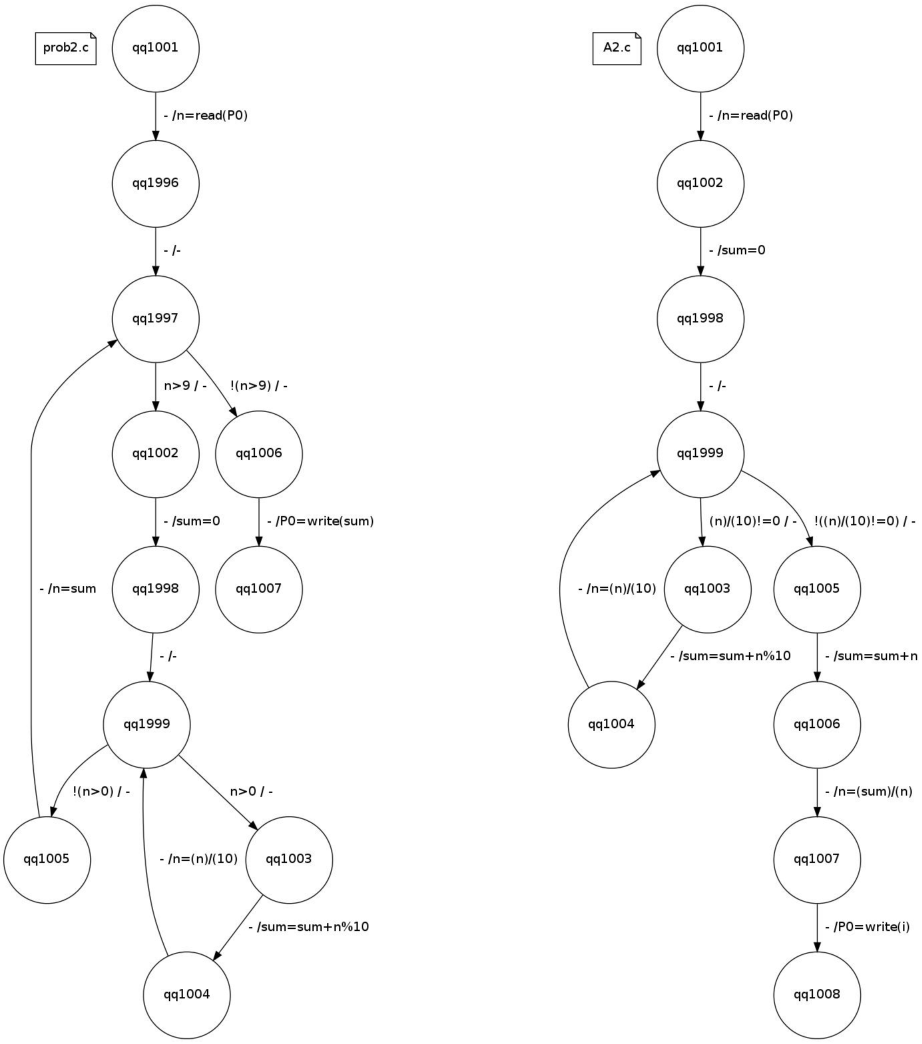
III. Program may contain uninitialized variables and have wrongly displayed output

Program for the above condition is given as follows:

```
#include <stdio.h>
//Problem 2
//Goes on adding the sum of digits until a single digit result is
//found, e.g.: 12345 --> 1+2+3+4+5 = 15 --> 1+5 = 6
int main() {
   unsigned int n, sum;
   printf("\n Input number:");
   scanf("%d", &n);
   while (n > 9) {
        sum = 0;
        while (n > 0) {
           sum = sum + n % 10;
           n = n / 10;
        }
   n = sum;
   }
   printf("\n %d", sum);
   return 0;
}
```

```
#include<stdio.h>
int main() {
   int n, i, sum;
   printf("enter the number");
   scanf("%d", &n);
   sum = 0;
   do{
      sum = sum + (n % 10);
      n = n / 10;
   } while (n / 10 != 0);
   sum = sum + n;
   n = sum / n;
   printf("the sum is : %d", i);
}
```

The FSMDs are given as in the figure 3.24.



(a) (b)

Figure 3.24: Two FSMDs: Case III, variable mapping

Evaluation steps

1. In path 0 of P1 and P2 we find $n \mapsto n$ and $sum \mapsto sum$. So, no further checking is required.

2. Then we check output variables, which are *sum* and *i*, so we can say that wrong output has been displayed.

3.3.5 Results

The results of variable mapping between various pairs of programs belonging to each of the different cases are given in this section. Table of results for each class has been prepared by running the variable mapping algorithm. The tables are given below.

Code Name	*#Variable*	*Code Name*	*#Variable*	*#Mapped*
Table.c	3	Table1.c	3	0
V1.c	3	V2.c	3	3
max3.c	3	max3_1.c	3	3
Table.c	3	Check_Alphabet.c	1	1
Table.c	3	Factorial.c	3	2
Factorial.c	3	V1.c	1	1
Max3.c	3	Check_Alphabet.c	1	1
Max3.c	3	N_Sum.c	3	1
Table.c	3	N_Sum.c	3	2
fct.c	3	v1.c	3	1

Table 3.4: Condition not matched

In the table 3.4 “Condition not matched”, means that some condition of a conditional construct in one FSMD does not match with the condition of the conditional construct in the other FSMD.

In the tables 3.4 to 3.9 we chose some random programs. The corresponding program for each program of column #1 appear in column #2; these were modified to test the working of variable mapping algorithm. In table 3.4 we have presented the

Code Name	***#Variable***	***Code Name***	***#Variable***	***#Mapped***
Swap.c	3	Swap1.c	4	4
Swap.c	3	Swap2.c	2	3
Swap1.c	4	Swap2.c	2	4
Factorial.c	3	V1.c	1	1
N_Sum.c	3	V1.c	1	1
Swap1.c	4	Check_Alphabet.c	1	1
Swap1.c	4	Factorial.c	3	1
N_Sum.c	3	Swap1.c	4	2
Swap.c	3	Check_Alphabet.c	1	1
Swap.c	3	Factorial.c	2	1

Table 3.5: More variables

Code Name	***#Variable***	***Code Name***	***#Variable***	***#Mapped***
Abd.c	5	Abd.c.c	5	4
Max3.c	3	Max31.c	3	3
sumd.c	3	SumD.c	3	2
Swap.c	3	Sw.c	2	3
Swap1.c	4	Sw1.c	4	4
Table.c	3	Tab.c	3	2
Table1.c	3	Tab2.c	4	3
V1.c	3	V2.c	3	3
z1.c	3	z2.c	3	2
mx.c	4	mx1.c	4	4

Table 3.6: Different output

Code Name	*#Variable*	*Code Name*	*#Variable*	*#Mapped*
Abd.c	5	Abd.c.c	5	4
Max.c	3	Max3.c	3	3
mx.c	3	Max31.c	3	3
Palindrom.c	4	Pndrm.c	4	3
sumd.c	3	sm.c	3	2
sw.c	3	Swap.c	3	3
Table.c	3	Tab.c	3	2
V1.c	3	Tab.c	3	1
pel2.c	4	Pndrm.c	4	3
pel.c	4	Pndrm.c	4	3

Table 3.7: Wrong output

Code Name	*#Variable*	*Code Name*	*#Variable*	*#Mapped*
Ams.c	5	Amstrng.c	5	1
dia.c	4	diamond.c	4	3
ams2.c	5	Amstrng2.c	5	1
ams3.c	6	Amstrng2.c	5	1
ams4.c	6	Amstrng2.c	5	1
ams5.c	6	Amstrng2.c	5	3
ams5.c	6	Amstrng3.c	5	3
dia1.c	4	Diamond.c	4	4
dia2.c	4	Diamond.c	4	4
dia2.c	4	Diamond2.c	4	4

Table 3.8: Missing loop

Code Name	*#Variable*	*Code Name*	*#Variable*	*#Mapped*
Swap.c	3	SwU.c.c	3	1
Table.c	3	Tab.c	3	2
z1.c	3	z2.c	3	2
UnV.c	3	UnV1.c	3	2
v1.c	3	v2.c	3	2
sw.c	3	Swap.c	3	3
sw1.c	6	Swap1.c	6	3
tab.c	5	tbl.c	5	2
tab1.c	5	tbl1.c	5	5
v11.c	3	v12.c	3	3

Table 3.9: Uninitialized variables

variable mapping in programs in which some condition does not match. For example, in table 3.4, the program Table.c has its corresponding program Table 1.c, which has been structured so that variable mapping algorithm will target the code part which is most related to handle situations, where conditions do not match. All other tables were similarly made. Respective number of variables in both the codes and number of variables mapped have also been shown in the tables.
Some of the codes were indeed included in final result processing. We have used extra sets of codes, including these for automated marking.
The integration of variable mapping with equivalence and containment checking remains as our future work, which we have mentioned in the Chapter 5.

Discussion

The following has been observed from the results of our work in this chapter.

1. The suggested approach is useful in ensuring the correct order of precedences of conditions in the if-else if constructs in the programs as suggested in [81].

2. The algorithm developed for variable mapping has been successfully tested on various programs having different variable names.

3.4 Conclusion

The growing number of students in academic institutions has naturally raised a concern regarding fast and consistent evaluation of student submissions. Research on designing automated program evaluators has received impetus in recent years to meet this growing demand. A construct that appears frequently in programs is the `else-if` block. While writing programs it has to be ensured that the condition that appears in a subsequent `else-if` block is not weaker than any of its preceding conditions, otherwise this block will never be executed. However, wrong ordering of conditions in `else-if` blocks are often introduced by novice student programmers. It is to be noted that such violation of precedence of conditions in `else-if` constructs qualify as logical errors which are not detected by standard compilers, such as *gcc*. Existing automated evaluation schemes, such as [77], also fail to handle such errors. In this work, we have identified various cases of incorrect precedence of conditions in `else-if` constructs and have accordingly prescribed rules to correct such constructs. Our automated checker detects violation of precedence of conditions in `else-if` constructs in students' programs and also provides feedback towards error correction based on the supplied rules.

Thus, in this chapter, we discussed the pre-processing requirements for the student's programs, so that further evaluation can be done, using the corrected FSMD of the student's program. The corrected FSMD can be subjected to equivalence checking, modified with containment analysis as discussed in chapter 2. The problem of detecting proper ordering of conditional constructs was thus important to check for possibility of dead code in a student's program. Variable mapping was necessary, as the basic equivalence checking mechanism requires the programs being compared to have the same variable names. The results of our schemes for the two cases have established the importance of such an exercise. In future, we shall incorporate implementation for more such cases. Moreover, in our future work, we shall try to generalize the techniques and make them more versatile so that programs with larger number of features like pointers, user-defined data-types, etc. are also covered.

Another important aspect of program evaluation is the ability for equivalence checking of approximately or nearly equal expressions. Once such a scheme is in place, the task of automated evaluation can be taken up. These aspects provide the scope for the contents of the next chapter, where the goal eventually is to design an FSMD based automated marking scheme.

Chapter 4

Supporting techniques for checking and evaluation of students' programs

In this chapter we describe supporting techniques for checking and evaluation of students' programs such as *i)* checking equivalence of approximately equivalent expressions and *ii)* develop a marking scheme for the programming exercises. In the first part of this chapter we discuss about checking equivalence of approximately equivalent expressions. Comparison of approximately equivalent functions [78, 79] may be required as an aid to equivalence checking discussed earlier. This may be because equivalence checking is not able to determine such equivalences, where two expressions which are approximately equivalent are to be examined for equivalence. In this work, therefore, we have used various approaches including a randomised simulation based approach in which a function is examined at random points in the domain of the other function. The closeness in their values at most of the points may indicate their approximate equivalence. In the later part of this chapter we present a marking scheme for the programming exercises. an algebraic formula has been suggested to compute marks for the student's program, using some constants, which need to be empirically established.

4.1 Approximate equivalence checking of expressions

4.1.1 Introduction

In the earlier chapters it was assumed that the equivalence of expressions could be checked via normalization [75]. There are cases where this method of normalization does not work. Some of such cases have been discussed in this part of the chapter. A fundamental problem in computer science is to decide if two expressions are equivalent [28, 31, 34, 95]. Checking equivalence of two arbitrary functions is an undecidable problem. Both the forms of equivalence checking mentioned in the thesis, the path extension based and the value propagation based equivalence checking, demand canonical representation of expressions, in order to compare two given expressions. FSMD based equivalence checking uses a normalized form, which is for integer arithmetic [75] and it is not a canonical form. Integer arithmetic does not have canonical form. If it had a canonical form, it would be decidable, but we know integer arithmetic is undecidable. Although not unique, we still use normalization of [75], because it maximizes the possibility of two equivalent expressions becoming syntactically identical. Thus, normal form is not a canonical form, it is only an attempt to bring the different representations of an expression together. Whether they will be same or not, cannot be claimed. All equivalent expressions, therefore, do not become syntactically identical after normalization. The normal form is unique (i.e., canonical) for polynomials, so equivalence of two polynomials can be checked using FSMD based equivalence checking. Switching algebra expressions have canonical forms, such as the sum of products or the product of sums forms. In the absence of a canonical representation, due to commutative property, an expression having n terms can be represented in 2^n forms [95]. For the expressions which can be normalized, such as expressions involving algebraic polynomials, this problem of exponential number of possible representations may be handled, as all the forms will have the same normal form. Those expressions, for which normalization cannot be done, or those expressions whose various representations have different normal forms, various ways of representing the same expression gives rise to problem in establishing their equivalence. Equivalence checking of expressions, both algebraic as well as transcendental, e.g., trigonometrical identities, is thus not trivial, because there is no known canonical form of representation for them. In case of algebraic polynomial fractions, we know they can be decomposed into partial fractions. The fractions before decomposition and after decomposition have different forms, and their normalized form may not be unique. FSMD based equivalence checking will not be possible due to the existence of different normal forms. That is to say that in the absence of a canonical form, their equivalence cannot be established by representing them in some common form of representation in an FSMD. Hence, there is a need for establishing their equivalence by computing their values at various points of interest and checking that the values of equivalent expressions are equal at those points. In case the values at all points could be checked to be equal, it could be said that one expression overlays the other. Doing this, however, is not possible as this may require evaluation at uncountable number of points. The equivalence, thus, can only be established in a probabilistic sense, which means that if the evaluation of two equivalent expressions at a very large number of points results in equal values at all the points, then with a high confidence we can say that they are equivalent. We, thus, cannot claim the equivalence with 100% confidence, yet a high confidence value may be achieved by taking a very large number of sample points. We can, therefore, rely upon this way of randomised simulation to achieve equivalence. Checking for approximate equivalence may be motivated by the following discussion. Apparently, floating point numbers can be treated in the same way as real numbers or bit-vectors; however, it is not so. In usual mathematics, addition and multiplication of real numbers (and bit-vectors) obeys law of associativity. By contrast, in computer science, the addition and multiplication of floating point numbers is not associative, as rounding errors are introduced when dissimilar-sized values are joined together. This is illustrated by an example [1]. It may be noted that techniques exist to minimize rounding errors, such as, Kahan summation algorithm [35, 42]. In such cases checking for approximate equivalence may be useful, where the normal forms are not equivalent, yet the values computed for the two expressions are approximately equal. In the cases where normalization is not possible, yet the values computed for the two expressions are approximately equal, the checking for approximate equivalence is useful.

Approximate equivalence of expressions has been aimed only at those expressions,

[1] , consider a floating point representation with a 4-bit mantissa:
$(1.000_2 \times 2^0 + 1.000_2 \times 2^0) + 1.000_2 \times 2^4 = 1.000_2 \times 2^1 + 1.000_2 \times 2^4 = 1.001_2 \times 2^4 \ldots$ (i)
$1.000_2 \times 2^0 + (1.000_2 \times 2^0 + 1.000_2 \times 2^4) = 1.000_2 \times 2^0 + 1.000_2 \times 2^4 = 1.000_2 \times 2^4 \ldots$ (ii)
Clearly, the values computed by equations (i) and (ii) are different even though they involve the same operands.

where canonical forms are not achievable through normalization, as FSMD based equivalence checking will not be possible in absence of a canonical form, due to existence of different normal forms. In such cases, checking for approximate equivalence may be useful, where the normal forms are not equivalent, yet the values computed for the two expressions are approximately equal. In the cases where canonization using normalization is not possible, yet the values computed for the two expressions are approximately equal, the checking for approximate equivalence is useful. This method is useful in the cases where algebraic simplification, which is an incomplete procedure, is needed to establish equivalence. Supporting examples are given in the thesis. In this method of establishing equivalence, no algebraic simplification is required.

4.1.2 Example of approximate equivalence checking

We consider the following well known identity

$\sin 2\theta = 2 \sin\theta \cos\theta$

The LHS of the identity can be treated as one function and the RHS as the other. The range of θ for evaluation can be conveniently chosen from 0 to π.

We can start with choosing 10 points randomly, generated using rand function, the values of the two functions can be computed at those points and their difference is also computed at each point. At all the points there will be no difference in the computed values of the expressions. This suggests that at all these points, the functions may be equal.

4.1.3 Equivalence checking with randomised simulations with some known properties

The *equality* of two functions has been defined as follows [36]: “Two functions $f : A \rightarrow B$ and $g : A \rightarrow D$ are equal if $f(x) = g(x)$ for every $x \in A$.”

The co-domains of f and g need not be the same in order that $f = g$ holds. For the functions $f : \mathbb{Z} \rightarrow \mathbb{N}$ and $g : \mathbb{Z} \rightarrow \mathbb{Z}$ defined as $f(x) = \mid x \mid +2$ and $g(x) = \mid x \mid +2$, even their co-domains are different, the functions are equal because $f(x) = g(x)$ for every x in the domain.

In actual engineering problems, where floating point numbers and expressions on those are used, often some approximate solutions, within an acceptable error, are admissible. Thus, in student's program a student may use an expression which evaluates approximately equal to the values yielded by the expression in the golden program. In order to check the approximate equivalence in such cases, we need to define approximately equal functions, which evaluate approximately equal values, within acceptable error. Such functions will not have the same normal form and are thus to be dealt with differently for checking their approximate equivalence. Approximate equality of functions is to be checked where checking by way of normal forms of expressions will not work, on expressions over floating point variables. Thus checking via approximate equality is to be used in lieu of checking via normalization in such cases.

To start with, we define approximately equal univariate functions within a given range as follows:

Definition 16 (Approximately equal functions over a range)**.** *Let $f_1 : A \rightarrow B$ and $f_2 : A \rightarrow B$ be two univariate functions. Let $\forall x \in A$, $f_1(x) = f_2(x) + \varepsilon$, $\mid \varepsilon \mid > 0$, where $x_1 \leq x \leq x_2$. We call f_1 and f_2 as approximately equal within ε in the range from x_1 to x_2 of the variable x. The range $x_1 \ldots x_2$ is known as the range of approximate equality (within ε) for the functions f_1 and f_2.*

According to our notion of approximate equality as stated above, it may be easier to automatically establish approximate equality in case of polynomials, if the roots of the polynomial are known. In cases, where the range of interest is given, the approximate equality may be established by evaluating the two functions at a large number of randomly chosen points in the range. In case of a periodic function, we should empirically check whether the other function is also periodic and has values same as the golden function, within ε at a large number of randomly chosen points from the range. From periodicity, the range can be manually found out for the the function in the golden program. We give below some examples of establishing approximate equality of functions (in these examples the functions in the golden program are such that their normalization is not possible). It will be observed that the checking procedure is not symmetric, as the instructor has the advantage of annotating the golden program with additional information to facilitate the checking.

Example 4.1. Consider checking the expression $(x+1.5)(x+2)$, in the golden program, denoted as $expression_g$, and the expression $x^2+3.5x+2.99$, in the student program, denoted as $expression_s$. The roots of $expression_g$ are x= -1.5, -2, as it is a polynomial of degree 2, in the factored form. $expression_s$ is also a polynomial of degree 2, which evaluates nearly to 0, at the roots of $expression_g$, suggesting that the two polynomials have nearly the same roots and degree. Thus, $expression_g$ is approximately equal to $expression_s$ within the given error bound $\varepsilon = .01$(say).[by the factor theorem [73]]

Steps of the computational procedure

Step 1: Check the nature of expressions and find the roots of the expression in golden program

If both the expressions in the pair $\langle expression_g, expression_s \rangle$ are polynomials, check if the $expression_g$ is in the form of a factored polynomial. Check that $expression_g$ and $expression_s$ are of the same degree. Use the roots of the factored form polynomial $expression_g$.

Step 2: Check if the expression in the student's program also vanishes at the roots of the expression in golden program

For all roots of $expression_g$, if $\Big| expression_g - expression_s \Big|_{roots} \leq \varepsilon$, then both $expression_g$ and $expression_s$ are approximately equal within ε.

Example 4.2. Consider the expression having only one term, $\sin 2\theta$ in the golden program. Let us denote it as $expression_g$. The student program may have $2\sin\theta\cos\theta$ as the corresponding expression, let us denote it as $expression_s$. Note that the $expression_g$ is periodic with period of $\theta = \pi$, which is manually annotated in the golden program. Finding that the $expression_s$ is also periodic with the same period, is done by evaluating the $expression_s$ at a large number of randomly chosen points from the period of $expression_g$, i.e., from 0 to π. It is observed that the $expression_s$ evaluates approximately equal, within ε, to the values of the function at the randomly chosen values of θ, from the period. Thus the $expression_s$ may be inferred to be approximately equal, within ε, to the $expression_g$.

Steps of the computational procedure

Step 1: Check the nature of expressions and the periodicity of the expression in golden program

Let $\langle expression_g, expression_s \rangle$ be the corresponding pair of expressions to be checked for approximate equality, check that $expression_g$ is periodic. Manually annotate the periodicity of $expression_g$ with the period 0 to T.

Step 2: Check if the expression in the student's program also equals the expression in golden program in the same period

For n randomly chosen samples of $x \in [0 \dots T]$, if $\left| expression_g - expression_s \right|_x \leq \varepsilon$, where n is a large number annotated manually in the golden program, then both $expression_g$ and $expression_s$ are approximately equal within ε. For each of the previously chosen n random samples of $x \in [0 \dots T]$, let $x = x + T : x \in [T \dots 2T]$. To check the periodicity of the $expression_s$, if $\left| expression_g - expression_s \right|_{x \in [T \dots 2T]} \leq \varepsilon$, then both $expression_g$ and $expression_s$ are approximately equal within ε and $expression_s$ is also periodic with the same period as that of $expression_g$.

To find the approximate location of a root, we can make use of the Matijasevik's theorem [84], which states that:

Given a polynomial $c_1 x^n + c_2 x^{n-1} + \dots + c_n x + c_{n+1}$ with a root at $x = x_0$. Let c_{max} be the largest absolute value of a c_i and c_1 be the coefficient of the higher order term, then $|x_0| < (n+1)\frac{c_{max}}{|c_1|}$, i.e., the roots of the polynomial must lie between the values $\pm k \frac{c_{max}}{c_1}$, where k is the number of terms in the polynomial.

Matijasevik's theorem shows that calculating such bounds for multivariate polynomials is impossible [84].

Approximate equality checking for functions in x in the golden program, with another function in x in the student's program.

Example 4.3. Consider the expression $\dfrac{1}{1-x}$, $|\, x \,| < 1$, in the golden program. The student may employ the series expansions to approximately evaluate the algebraic expression as $1 + x^2 + x^3 + x^4 + \dots$, $|\, x \,| < 1$.

He may approximate the infinite series by retaining the terms upto some degree in their respective series. Thus we may have the following expressions. $expression_g =$

$\frac{1}{1-x}$, $|\,x\,| < 1$, the expression in the golden program and $expression_s = 1 + x^2 + x^3 + x^4$, $|\,x\,| < 1$, the expression in the student's program.

When evaluated at various randomly chosen values of x in the range $-1 < x < 1$ (given), we will find that the values of $expression_g$ and $expression_s$ are close to each other. We can thus establish that $expression_g$ and $expression_s$ are approximately equal within a permissible error of ε.

Steps of the computational procedure

Step 1: Check the nature of expressions and for the function in x in the golden program, manually annotate the *range* of x

Let $\langle expression_g, expression_s \rangle$ be the corresponding pair of expressions to be checked for equality, check that $expression_g$ is not a polynomial in factored form and is not a periodic function.

Annotate manually the *range* of variable x of $expression_g$.

Step 2: Check if the expression in the student's program also equals the expression in golden program at the chosen number of random samples

For all $x \in range$, if $\left|expression_g - expression_s\right|_x \leq \varepsilon$, where x is chosen randomly n number of times, n being a large number annotated manually in the golden program, then both $expression_g$ and $expression_s$ are approximately equal within ε.

Approximate equality checking for trigonometric expressions using their power series expansions

In case the LHS and RHS of a trigonometric identity can be reduced to approximate polynomials, then we can establish the equality of the RHS and LHS by evaluating the polynomials at sufficiently large number of points compared to the degree of polynomial. We take an example below:

Example 4.4. Consider the expression $\frac{1-\tan x}{1+\tan x}$ in the golden program.
We explore the following possibilities of expressions written by the student in his program. The student may write either the form same as above or may write the following identical expression $\frac{\cos x - \sin x}{\cos x + \sin x}$. The student may also employ the Maclaurin series

expansions to approximately evaluate the trigonometric functions. The trigonometric ratios $\sin x$, $\cos x$ and $\tan x$ may be represented by their infinite series expansions as follows.

$$\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \ldots \text{ for every } x$$

$$\cos x = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \ldots \text{ for every } x$$

$$\tan x = x + \frac{x^3}{3!} + \frac{2x^5}{15} + \ldots \text{ for } |x| < \pi/2$$

He may approximate the infinite series by retaining the terms up to the fifth degree in their respective polynomials. Thus the following expansions result for the expression in the golden program and the expression $\frac{\cos x - \sin x}{\cos x + \sin x}$. The series expansions for the expression in the golden program

$$= \frac{(1 - x - \frac{x^3}{3!} - \frac{2x^5}{15})}{(1 + x + \frac{x^3}{3!} + \frac{2x^5}{15})}$$

and the series expansion for the expression $\frac{\cos x - \sin x}{\cos x + \sin x}$

$$= \frac{(1 - x - \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} - \frac{x^5}{5!})}{(1 + x - \frac{x^2}{2!} - \frac{x^3}{3!} + \frac{x^4}{4!} + \frac{x^5}{5!})}$$

For $x = 0$ all the above expressions evaluate to 1.

When evaluated at various values of $0 < x < \pi/2$, we will find that the values of all the above expressions are close to each other. We can thus establish that they are equivalent within a permissible error of ε.

In the following we consider an example of approximate equivalence checking, in which the golden program and the student's program have a statement which is equivalent, but cannot be normalized. We note that the equivalence checker finds two paths equivalent, if the conditions and the data-transformations on both match.

Example 4.5. Let us consider the two FSMDs given in figure 4.1. In the figure, the conditions and data-transformations along the transitions are shown as *cond* and *dtrans* with appropriate subscripts. It may be noted that some of the conditions and data-transformations are identical in M_s and M_g. The data-transformations are assumed to be in the normalized form, except for one in each of the FSMDs, which cannot be normalized and it's equality checking will be explained further. The cut-point to cut-point paths are shown as p_s and p_g with appropriate subscripts. The equivalence checking starts with the respective starting states, q_a and q_{01}, assumed to be corresponding states. Starting from this pair of corresponding states, the paths to the next cut-point in both the FSMDs are checked and found to be equivalent, i.e., $q_a \rightarrow q_b \cong q_{01} \rightarrow q_{02}$. Now q_b and q_{02} become corresponding states. The path $q_b \rightarrow q_c$ (corresponding transition $p_{s,1}$ in M_g) is checked next and found to have an equivalent path $q_{02} \rightarrow q_{03}$ ($p_{g,1}$ in M_g). The other path from q_b viz., $q_b \rightarrow\ q_c$ ($p_{s,2}$ in M_s) is found to have an expression $x = 2\sin\theta\cos\theta$, which cannot be normalized. Hence the equivalence checking for this expression is to be done to ascertain approximate equality. The expression in the next path to be checked in M_g, viz., the path $q_{02} \rightarrow q_{03}$ ($p_{g,2}$), is $\sin 2\theta$, which is annotated to be a periodic expression in the golden program with the period of θ from 0 to π. Hence, the approximate equality checking is done for the expressions $expression_g = \sin 2\theta$ and $expression_s = 2\sin\theta\cos\theta$ as per the steps given in the computation procedure for approximate equality of expressions when the golden program has a periodic function. The two expressions are evaluated at an annotated number of random sample points for θ in the range $[0\ldots\pi]$, in the golden program and the difference in the values is checked to be within an already annotated ε. The periodicity of $expression_s$ is established by repeating the trial in the range $\theta = \pi$ to 2π. As all the values are found within ε, the approximate equality of $expression_s$ is established with $expression_g$, resulting in equivalence of the paths $p_{s,2}$ and $p_{g,2}$. Thus, q_c and q_{03} become corresponding states. Let the next path chosen from q_c be $q_c \rightarrow q_d$ ($p_{s,3}$ in M_s) which is found to have the equivalent path $q_{03} \rightarrow q_{04}$ ($p_{g,3}$ in M_g). The next path from q_c is $q_c \rightarrow q_d$ ($p_{s,4}$ in M_s). This path is found to be equivalent to the path $q_{03} \rightarrow q_{04}$ ($p_{g,4}$ in M_g). The states q_d and q_{04} now become corresponding states. As no more paths remain to be checked, the two FSMDs are asserted to be one way equivalent. Now the equivalence of paths in both the FSMDs is also checked by reversing the roles of the FSMDs M_g and M_s. It will be seen that the FSMDs will again be found to be equivalent as the paths are exactly the same. Hence the FSMDs are equivalent both ways.

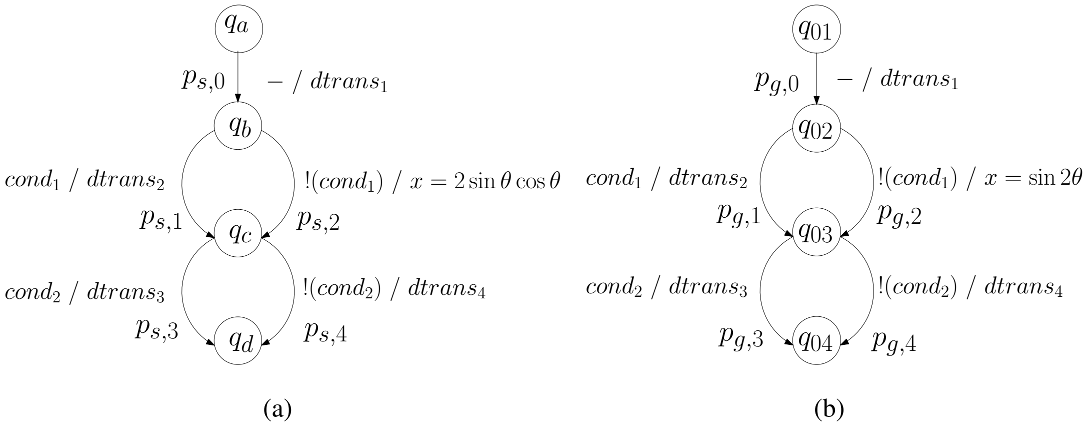


Figure 4.1: FSMDs (a) M_s and (b) M_g.

In the following we present an example of containment checking, which involves the approximate equality checking of functions. We note that in containment checking, a path p_s is said to be contained in another, say p_g, if the data transformations in p_s are also present in p_g, irrespective of their conditions.

Example 4.6. We consider parts of two FSMDs M_g and M_s as given in figure 4.2 from LCS and CSLCS onwards. In the figure, the conditions and data-transformations along the transitions are shown as *cond* and *dtrans* with appropriate subscripts. It may be noted that some of the conditions and data-transformations are identical in M_s and M_g. The data-transformations are assumed to be in the normalized form, except for one in each of the FSMDs, which cannot be normalized and it's equality checking will be explained further. Those conditions and data-transformations, which are not the same, have been shown with different subscripts. The cut-point to cut-point paths are shown as p_s and p_g with appropriate subscripts.

The containment checking proceeds as follows. The containment checker tries to find containment of the path $p_{s,1}$, i.e., $q_b \rightarrow q_c$ of M_s, which is from LCS to the next cut-point, inside the path $p_{g,1}$ i.e., $q_{02} \rightarrow q_{03}$ of M_g, which is from CSLCS to the next cut-point, as per the following discussion.

The path from q_b viz., $q_b \rightarrow\ q_c$ ($p_{s,1}$ in M_s) is found to have an expression $x = 2\sin\theta\cos\theta$, which cannot be normalized. Hence the equivalence checking for this expression is to be done to ascertain approximate equality. The expression in the path to be checked in M_g, viz., the path $q_{02} \rightarrow q_{03}$ (the path $p_{g,1}$, is $\sin 2\theta$, which is annotated to be a periodic expression in the golden program with the period of θ from 0 to π. Hence, the approximate equality checking is done for the expressions

$expression_g = \sin 2\theta$ and $expression_s = 2\sin\theta\cos\theta$ as per the steps given in the computation procedure for approximate equality of expressions when the golden program has a periodic function. The two expressions are evaluated at a large number of random points in the range of θ, which is already annotated in the golden program and the difference in the values is checked to be within an already annotated ε. The periodicity of $expression_s$ is established by repeating the trial in the range $\theta = \pi$ to 2π. As all the values are found within ε, the approximate equality of $expression_s$ is established with $expression_g$, resulting in equivalence of the paths $p_{s,1}$ and $p_{g,1}$. Thus, q_c and q_{03} become corresponding states.

As the sets of data-transformations along the two paths as above are now found to be the same, hence containment is asserted. The containment checker reports $p_{s,1}$ "Unordered path-wise both way contained" in path $p_{g,1}$. Both these paths are now excluded for further checking by the containment checker. Now the containment checker backtracks from q_{03} to q_{02} and tries to find the containment of the remaining path $p_{s,2}$ from LCS, in the remaining path $p_{g,2}$ emanating from CSLCS, i.e., q_{02}. This time containment is not found due to different data-transformation expressions. The path $p_{g,2}$ is, therefore, extended to the next cut-point q_{05} along one of the paths emanating from q_{03} , say $p_{g,3}$. As none of the data-transformations along the path $p_{g,3}$ is same as that of $p_{s,1}$, hence, the containment is not found. The containment checker now backtracks to the state q_{03} to explore the other path emanating from it, by extending along $p_{g,4}$. This time again the containment is not found for the similar reason as before. The containment checker, therefore backtracks to q_{03}, and as there is no other path from it, hence it further backtracks to q_{02}. As all the paths from q_{02} have been explored and as nowhere the containment was found, hence the containment checker reports $p_{s,2}$ as "path-wise un-contained".

In the following we present an example of checking and correcting the error in a student program using approximate equivalence checking of the FSMDs of student's program and the golden program.

Example 4.7. The FSMDs of the student's program and the golden program are shown in figure 4.3. We note that the equivalence checker finds two paths computationally equivalent, if the conditions and the data-transformations on both match. We also note that in containment checking, a path p_s is said to be contained in another, say p_g, if the data transformations in p_s are also present in p_g, irrespective of their

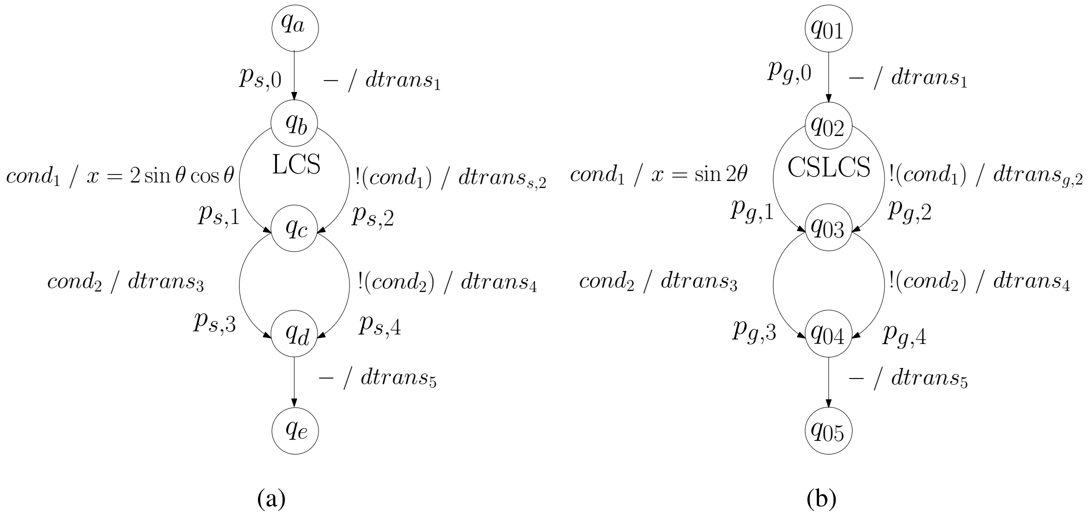


Figure 4.2: FSMDs (a) M_s and (b) M_g.

conditions. The checking and correction process is executed in the following steps.

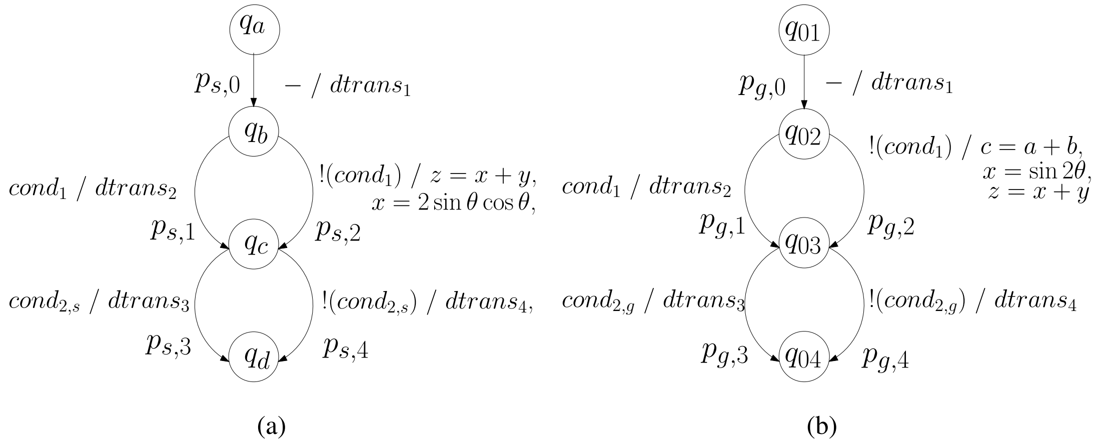


Figure 4.3: FSMDs (a) M_s and (b) M_g.

- Equivalence checker is executed in the beginning. It finds the path $q_a \rightarrow q_b$ (path $p_{s,0}$ in M_s), equivalent to the path $q_{01} \rightarrow q_{02}$ (path $p_{g,0}$ in M_g), as they have the same condition (-) and data-transformation ($dtrans_1$). q_b and q_{02} become corresponding states.

- Equivalence checker then finds the equivalence of the paths $q_b \rightarrow q_c$ (path $p_{s,1}$ in M_s), equivalent to the path $q_{02} \rightarrow q_{03}$ (path $p_{g,1}$ in M_g), as they have the identical condition of execution ($cond_1$) and identical data-transformation ($dtrans_2$).

- After this, the equivalence checker fails to find the equivalence of the path $q_b \rightarrow q_c$ (path $p_{s,2}$ in M_s), with the path $q_{02} \rightarrow q_{03}$ (path $p_{g,2}$ in M_g), as they have the same condition of execution, $!(cond_1)$, but not the same set of data-transformations along the two paths, even after doing path extensions $p_{s,2}$ with $p_{s,3}$ and also $p_{s,2}$ with $p_{s,4}$. Moreover the data-transformations, $x = 2\sin\theta\cos\theta$ in M_s and $x = \sin 2\theta$ in M_g, cannot be normalized, for which approximate equality checking is used, before equivalence checker fails.

- As a result, q_b becomes LCS and q_{02} becomes CSLCS, from where the containment checker starts checking. It finds $p_{s,1}$ contained in $p_{g,1}$, as the data-transformation along them are identical ($dtrans_2$).

- The containment of $p_{s,2}$ is then found in $p_{g,2}$, as the set of data-transformations in $p_{s,2}$ is a subset of the set of data-transformations in $p_{g,2}$. Here also, as the data-transformations, $x = 2\sin\theta\cos\theta$ in M_s and $x = \sin 2\theta$ in M_g, cannot be normalized, for which approximate equality checking is used.

- As the correction step, the extra statement statement $c = a + b$ in $p_{g,2}$ is copied to $p_{s,2}$. As the equivalence checker failed but containment checker succeeded, so the two dependent data-transformations are interchanged in the $p_{s,2}$, making the two paths equivalent, which will be seen in the next step.

- Again, equivalence checker is executed to check the extent of equivalence of the two FSMDs. This time equivalence checker is able to find equivalence of both the paths $p_{s,1}$ and $p_{s,2}$ emanating from q_b and meeting at q_c with the paths $p_{g,1}$ and $p_{g,2}$ emanating from q_{02} and meeting at q_{03}. Thus q_c and q_{03} become corresponding states.

- Equivalence checker starts to explore the paths emanating from the recently established corresponding states q_c and q_{02}, but it fails to check further due to mismatch in the conditions of execution. Thus q_c becomes new LCS and q_{03} becomes new CSLCS.

- The containment checker is now called and it starts finding containment of path $p_{s,3}$ from LCS to q_d in the path $p_{g,3}$ from CSLCS to q_{04}. The data-transformation being the same along the two paths, containment is asserted. As the conditions are different, so the condition of execution of the path $p_{s,3}$ is replaced with the condition of the path $p_{g,3}$ of golden program.

- The containment checker now checks containment of path $p_{s,4}$ from LCS to q_d in the path $p_{g,4}$ from CSLCS to q_{03}. After finding the containment the condition of execution of the path $p_{s,4}$ is made same as that in the golden program, as done in previous step.

- Finally the equivalence checker is executed to establish that both the FSMDs are equivalent.

In the above example we have observed the following:

1. $p_{s,1} \simeq p_{g,1}$ is also checked by the equivalence checker in its first run, before it failed, but this check is not reported. The containment checker again checks it.
2. If containment is found, the order of statements can still be different. So the order is to be restored as per the golden program.
3. As correcting step, the missing statements in the student's program are copied from the golden program, in the order in which they occur in the golden program. Also the condition of execution in the golden program at the cut-point state, is copied to the student program as a correcting step.

Conclusion, discussion and future work:
We may thus arrive at a notion of the ε-cover of a given expression.

Definition 17 (ε-cover). *ε-cover of an expression may be defined as the function whose value when evaluated differs by at most ε, from the value of the given expression, at a large fraction of points chosen at random.*

Definition 18 (Probably equivalent expressions). *If two expressions lie within ε-cover, then they are said to be* **probably equivalent** *within the permissible error of ε.*

Our problem is that we want to be able to state the probable equivalence with high confidence value, and find out what should be the number of samples to be chosen for this. As it turns out, the relation of sample size with confidence interval may be given by Chernoff bound. In doing so we would also like to use as less sampling points as possible.

Chernoff bound approach for finding minimum number of samples in checking approximate equality of functions is given below with an example.

In order to check the approximate equality of two functions, say $f_1(x)$ and $f_2(x)$, in a given interval, say $[x_1, x_2]$, we may have two cases. In case the functions are approximately equal then the two given functions have approximately equal values at the most of uniformly distributed samples in the interval $[x_1, x_2]$, i.e., $\mid f_1(x_i) - f_2(x_i) \mid \leq \varepsilon$, ε being the maximum allowable error. In case the functions are not approximately equal, they will differ significantly, i.e., $\mid f_1(x_i) - f_2(x_i) \mid \geq \varepsilon$, at most of the points x_i in the interval $[x_1, x_2]$. It will be of interest to know, in order to minimize the computational effort, what is the minimum number of sample points, which will be sufficient to establish the approximate equality of two given functions with a high confidence value. The minimum number of samples is to be determined for the two cases, the first being the case that $f_1(x)$ and $f_2(x)$ are approximately equal and the other in which the two functions are not approximately equal. We discuss below the Chernoff bound approach for the first case, the approach for the other case may be taken similarly.

Let $\varepsilon \in (0,1)$ be the margin of absolute error and $\delta \in (0,1)$ be the confidence parameter. To determine the approximate value of the error bound, we have the following expression

$$Pr(|\tilde{L_n} - L| \leq \varepsilon) \geq 1 - \delta \tag{4.1}$$

where L is the probability that $\mid f_1(x_i) - f_2(x_i) \mid \leq \varepsilon$, where x_i is chosen randomly from $[x_1, x_2]$ and $\tilde{L_n}$ is used to denote the output of a random sampling (for estimating L). Using the above formula, we can use Chernoff bound [8, 16, 23, 62, 82, 92], which is used extensively in computer science. In the random sampling techniques, Chernoff bounds are used to determine the upper limit of the significant difference between two objective functions. By considering the Chernoff bound, we get more accurate bounds for tail probability of sum of Bernoulli random variables. Next, we state the Chernoff bound for average of the sum of Bernoulli random variables as follows.

$$Pr\left[\frac{X}{n} > (1+\varepsilon)p\right] \leq exp(-\frac{pn\varepsilon^2}{3}), Pr\left[\frac{X}{n} < (1-\varepsilon)p\right] \leq exp(-\frac{pn\varepsilon^2}{2}) \tag{4.2}$$

Using the bound given above, we can calculate several formulas for sample size. In the following, we give the formula to obtain the sample size to find approximate equivalence of two functions using the two equations 4.1 and 4.2 [92], where $X = \sum_{i=1}^{n} X_i$ and n is the minimum sample size. If sample size n satisfies one of the following inequalities, then it satisfies the bound (4.1).

$$n > \frac{1}{2\varepsilon^2} ln(\frac{2}{\delta}) \tag{4.3}$$

and

$$n > \frac{3L}{\varepsilon^2} ln(\frac{2}{\delta}). \tag{4.4}$$

Example 4.8. Application of Chernoff bound

As an example of application of Chernoff bound we have the following minimum sample sizes n, for various desired values of δ and ε, according to the inequality 4.4 given above, assuming $L = 1$ (most desirable case).

$\delta = 0.1$ $\varepsilon = 0.2$ $n \geq 225$

$\delta = 0.01$ $\varepsilon = 0.2$ $n \geq 397$

$\delta = 0.001$ $\varepsilon = 0.2$ $n \geq 570$

$\delta = 0.001$ $\varepsilon = 0.2$ $n \geq 742$

We observe from the above data that for the same error tolerance (ε), if the confidence parameter (δ) is reduced, the number of minimum samples required increases monotonically but slowly.

We next present the data for minimum sample size, where the confidence parameter (δ) is fixed, but the error tolerance (ε) is increased.

$\delta = 0.1$ $\varepsilon = 0.2$ $n \geq 225$

$\delta = 0.1$ $\varepsilon = 0.02$ $n \geq 22468$

$\delta = 0.1$ $\varepsilon = 0.002$ $n \geq 2246799$

We observe that for the same confidence parameter (δ), if the error tolerance (ε) is reduced, the number of minimum samples required increases rapidly.

Reducing the sample points may be possible in a case where e.g. if the reference function is well behaved, e.g., piecewise linear or piecewise smooth between corner points. Checking equivalence only at corner points may suffice, reducing the number of sample points. However as the reference function may be well behaved but as there may not be any knowledge about the function being compared, so it may be required to test at all points. In such a case where there is none but the only way to sample at a large number of points, we may need to look for randomly finding out the sample points, such that when tested over those points, the equivalence can be established with some high confidence.

Reducing number of sample points may be possible if we can reduce the given functions to some polynomials with some finite terms and if we can find the roots of the polynomials, then the points for evaluating the polynomials could be just at the roots. So for a degree n polynomial, as it has n roots, checking at n points could be sufficient, all the points being the roots of the polynomials.

4.1.4 Obtaining the range of evaluation from the conditions in the program

At times, it may be possible to obtain the range from the conditions given in the program. In case, the condition for evaluation is in sum of products form, each product term may yield a range of computation, as shown in the following example.

Example 4.9. Let there be following conditions in a given program in which the expression given below is valid.

$(x < 10 \;\&\&\; x > 5) \;||\; (x < 20 \;\&\&\; x > 15) \;||\; (x < 50 \;\&\&\; x > 40)$

$$\frac{(x-1)(x+1)}{x^2+1}$$

Let another program have the same conditions under which the same expression is written in a different form as follows.

$(x < 10 \;\&\&\; x > 5) \;||\; (x < 20 \;\&\&\; x > 15) \;||\; (x < 50 \;\&\&\; x > 40)$

$$\frac{(x^2-1)}{x^2+1}$$

That the expressions in the two programs are equivalent, is obvious from the basic algebra, but the equivalence checker cannot check their equivalence as their normal forms for the two expressions will not be the same. In such a case the equivalence may be suggested by actually evaluating the two expressions in the range obtained from the conditions given in the two programs and then comparing the values at every point in the range. If the difference in the values at every point is the same, then the two expressions may be equivalent. The range for evaluating the expression may be obtained from the conditions as values of x from 6 to 9, 16 to 19 and 41 to 49.

We bring another example from trigonometric identities as follows.

Example 4.10. Let there be following conditions in a given program in which the expression given below is valid.

$(\theta < \pi/2 \ \&\& \ \theta > 0) \ || \ (\theta < 3\pi/2 \ \&\& \ \theta > \pi)$

$$\frac{1 - \tan\theta}{1 + \tan\theta}$$

Let another program have the same conditions under which the same expression is written in a different form as follows.

$(\theta < \pi/2 \ \&\& \ \theta > 0) \ || \ (\theta < 3\pi/2 \ \&\& \ \theta > \pi)$

$$\frac{\cos\theta - \sin\theta}{\cos\theta + \sin\theta}$$

That the expressions in the two programs are equivalent, is obvious from the basic trigonometry, but the equivalence checker cannot establish their equivalence as the two expressions are syntactically not the same. In such a case the equivalence may be suggested by actually evaluating the two expressions in the range obtained from the conditions given in the two programs and then comparing the values at every point in the range. If the difference in the values at every point is the same, then the two expressions may be equivalent. The range for evaluating the expression may be obtained from the conditions as values of θ vary from 0 to $\pi/2$ and from π to $3\pi/2$.

4.1.5 The decision procedure

We present below the decision procedure in algorithm 17 to check the equivalence of two expressions, in case they are univariate and cannot be normalized or have different (not equivalent) normal forms. In algorithm 17, the check in line 3 is about deciding whether expressions can be normalized? Normalization is a well defined procedure. Normalization is a sound but not complete mechanism for checking equivalence of arithmetic expressions. The normalization scheme used here is adapted from [43].

In the golden program and the student program, *i)* C_g, C_s stand for the conditions in the path, *ii) expression*$_g$ and *expression*$_s$ stand for the expressions .

Steps for the decision procedure for equivalence checking in a range

A *range* [$start$, *end*] has two values, *i)* starting value of range, $start$ and *ii)* End value of range, *end*.

SOP (*range*$_1$ OR *range*$_2$ OR ... OR *range*$_n$)

(1) Let a structure pair have two fields

  (I) pair.first will hold start value of a range.

  (II) pair.second will hold end value of a range.

  Let SOP_pairs[] be the array to hold all pairs of a SOP.
  Let CG[][] be the array to hold all SOP_pairs of golden program M_g.
  Let CS[][] be the array to hold all SOP_pairs of student program M_s.

(2) Let stmt[] be the array to hold statements of a SOP.
  Let stmtG[][] be the array to hold all stmt of M_g.
  Let stmtS[][] to be the array to hold all stmt of M_s.

(3) Let ResultG[i][j] be the array to store result for i^{th} SOP condition and j^{th} statement of M_g.
  Let ResultS[i][j] be the array to store result for i^{th} SOP condition and j^{th} statement of M_s.

Steps are as follows:

(1) Parse the code

  (I) For all conditions C_i in M_g

    (A) If C_i is in SOP form Extract SOP_pairs[] of SOP for C_i and store at CG[i][].
    Extract all statements of C_i to stmt[] and store at stmtG[i][]

```
      (II) For all conditions $C_i$ in $M_s$
          (A) If $C_i$ is in SOP form Extract SOP_pairs[] of SOP for $C_i$ and store at CS[i][].
              Extract all statements of $C_i$ to stmt[] and store at stmtS[i][]
  (2) Compare SOPs and Results
      (I) For all i in CG[i][]
          (A) For all j in CS[j][]
              (i) If SOP_pairs[i] of CG[i][] is equal to SOP_pairs[j] of CS[j][]
                  (a) For all k of CG[i][k]
                      (1) For all l of CS[j][l]
                          (I) ResultG[i][k] = Evaluate statement stmtG[i][k]
                          (II) ResultS[j][l] = Evaluate statement stmtG[j][l]
                  (a) For all k of ResultG[i][k]
                      (1) For all l of ResultS[j][l]
                          (I) If ResultG[i][k] is equal to ResultS[j][l] then stmtG[i][k] is equal to stmtS[j][l]
```

4.1.6 Results

1. We did a simulation for the following well known identity using Matlab: $\sin 2\theta = 2\sin\theta\cos\theta$. The LHS of the identity was treated as one function and the RHS as the other. The range of θ for evaluation was chosen from 0 to π. To start with 10 points were chosen randomly, generated using rand function, the values of the two functions were computed at those points and their difference was also computed at each point. At all the points there was no difference in the computed values of the expressions. This proves that at all these points the functions are equal.

2. The results obtained by running the algorithm 17 for various expressions are given in the table 4.1. From the table it is evident that the method is capable of differentiating between the equivalent and non equivalent expressions.

Algorithm 17: Decision procedure for approximate equivalence checking

```
forall paths in FSMD do
    forall expressions in the path do
        if ( expression_g and expression_s cannot be normalized || expression_g
          and expression_s have not equivalent normal forms ) then
            if C_g and C_s are in sum of products form then
                Find the range of variable $x$ from each clause of $C_g$ (or $C_s$)
                forall x in the range do
                    if $(expression_g - expression_s)\big|_x < \varepsilon$ then
                        Output expressions may be equivalent
                    else
                        Output expressions are not equivalent
            else
                Ask for range of variable $x$ and number of samples $n$
                for 1 to n do
                    Pick $x$ randomly
                    if $(expression_g - expression_s)\big|_x < \varepsilon$ then
                        Output expressions may be equivalent
                    else
                        Output expressions are not equivalent
```

Expression1	***Expression2***	***Range***	***#Samples***	***Result***
$\frac{(x-1.1)(x+1.1)}{x^2+1.2}$	$\frac{x^2-1.21}{x^2+1.21}$	$*$	5	Equivalent
$\frac{x^2+1.2}{(x-1.1)(x+1.1)}, x \neq \pm 1.1$	$\frac{x^2+1.21}{x^2-1.21}, x \neq \pm 1.1$	$*$	5	Equivalent
$a \ \% \ 2$	$a \ \& \ 1$	$2 \leq a \leq 10$	5	Equivalent
$\sin 2\theta$	$2 \sin \theta \cos \theta$	0 to π	10	Equivalent
$\frac{1-\tan\theta}{1+\tan\theta}$	$\frac{\cos\theta-\sin\theta}{\cos\theta+\sin\theta}$	†	10	Equivalent
$\sin \theta$	$\cos \theta$	0 to π	10	Not Equiv.
`x=printf(str)`	`y=str.length()`	various strings	10	Equivalent
$\frac{(x^2+3.5x+2.99)}{(x^2-3.5x+3)}, x \neq 1.5, 2$	$\frac{(x+1.5)(x+2)}{(x^2-3.5x+2.99)}, x \neq 1.5, 2$	$*$	5	Equivalent
$\frac{(x-1.5)(x-2)(x-3)}{(x+1.5)(x+2)(x+3)}$	$\frac{(x^2-3.5x+3)(x-3)}{(x^2+3.5x+2.99)(x+3)}$	$*$	5	Equivalent
$\frac{x^2-1.21}{x^2+1.21}$	$\frac{x^2+1.21}{x^2-1.21}, x \neq \pm 1.1$	$*$	5	Not Equiv.

$* \ (x < 10 \ \&\& \ x > 5) \ || \ (x < 20 \ \&\& \ x > 15) \ || \ (x < 50 \ \&\& \ x > 40)$

$\dagger \ (\theta < 90 \ \&\& \ \theta > 0) \ || \ (\theta < 270 \ \&\& \ \theta > 180)$

Table 4.1: Checking expression equivalence

In the expression “a & 1” in the above table, the symbol & stands for bitwise AND operator. Also, as in the above table `x=printf(str)`, `y=str.length()`, in which case `x` and `y` will be equal.

4.2 Automated evaluation of programs

Automatically evaluating the student’s programs is a challenging task, because every student uses his own set of variable names and there may be several golden solutions to the problem.

A marking scheme for the programming exercises has been developed and has been tested to perform well. Based on FSMD equivalence checking, using propagation vectors, we have also suggested an improved method [77], which is given in section 4.2.4.

4.2.1 An automated program evaluation scheme

The task of evaluation module is to evaluate the student's program, based on the similarity with the golden model. For marking scheme the evaluation module uses two constants c_1 and c_2, which are empirically taken from the teacher, who based on his experience can provide them. The constant c_1 is used as a weight for number of variables successfully mapped and the constant c_2 is for the weight for the number of paths matched with the golden program. The total marks obtained by a student is thus the weighted sum of the proportions of the variables mapped and the paths matched. The complete formula for evaluation is as follows.

$$Marks = c_1 \times \frac{Totalnumberofvariablesmapped}{Totalvariablesinmodelprogram} + c_2 \times \frac{Totalnumberofpathsmatched}{Totalnumberofpathsinmodelprogram}$$

Whenever all the variables of student's program match with all variables of model program and all paths of the model program match with all the paths of student's program, the student is awarded full marks.

MappedVars is total number of variable matched between golden and student program by the function propagate.

Algorithm 18: Evaluation of student program

```
Function Evaluation (Pathlist1, Pathlist2, MappedVars, C1, C2)
pathMatched = 0
forall path i in golden program do
    forall path j in student program do
        if all statements of path i are there in path j then
            pathMatched ++
Marks = (MappedVars/TotalVar) * C1 + (pathMatched/Total paths in golden
  program) * C2
return
```

Code Name	***Marks Obt.***	***TA Marks***
1.c	5.1	5
2.c	6.9	7
3.c	9.3	9
4.c	8.7	9
5.c	8.1	8
6.c	8.7	9
7.c	9.3	9
8.c	8.7	9
9.c	6.1	6
10.c	5	5

Table 4.2: Automatic evaluation vs TA’s evaluation for Coordinate.c, C1=3, C2=7

Code Name	***Marks Obt.***	***TA Marks***
1.c	8.7	9
2.c	8	7
3.c	9.6	9
4.c	9.3	9
5.c	9.6	9
6.c	8.9	9
7.c	7.3	8
8.c	7.3	7
9.c	9.6	9
10.c	9.6	9

Table 4.3: Automatic evaluation vs TA’s evaluation for Friend.c, C1=3, C2=7

Code Name	***Marks Obt.***	***TA Marks***
1.c	9.3	9
2.c	8.1	8
3.c	10	10
4.c	10	10
5.c	8.9	9
6.c	8.9	9
7.c	6.5	7
8.c	4.6	5
9.c	8.1	8
10.c	7.3	7

Table 4.4: Automatic evaluation vs TA's evaluation for Equal0_1.c, C1=3, C2=7

Code Name	***Marks Obt.***	***TA Marks***
1.c	9.3	9
2.c	10	10
3.c	10	10
4.c	9.4	9
5.c	10	10
6.c	10	10
7.c	9.4	9
8.c	8.7	8
9.c	8.1	8
10.c	9.4	9

Table 4.5: Automatic evaluation vs TA's evaluation for GCDBenchmark.c, C1=3, C2=7

Code Name	***Marks Obt.***	***TA Marks***
1.c	9.0	9
2.c	7.6	8
3.c	10	10
4.c	8.6	9
5.c	7.2	7
6.c	4.4	5
7.c	5.3	6
8.c	10	10
9.c	10	10
10.c	2	2

Table 4.6: Automatic evaluation vs TA's evaluation for LCM.c, C1=3, C2=7

Code Name	***Marks Obt.***	***TA Marks***
1.c	8.0	8
2.c	8.6	9
3.c	7.9	8
4.c	8.6	9
5.c	8.6	9
6.c	10	10
7.c	10	10
8.c	7.4	8
9.c	1.8	2
10.c	7.4	8

Table 4.7: Automatic evaluation vs TA's evaluation for GCD.c, C1=3, C2=7

Code Name	***Marks Obt.***	***TA Marks***
1.c	6.7	7
2.c	6.7	7
3.c	6.8	7
4.c	7.3	8
5.c	7.2	7
6.c	7.2	7
7.c	7.2	7
8.c	7.1	7
9.c	5.6	6
10.c	7.2	7

Table 4.8: Automatic evaluation vs TA's evaluation for BarCode.c, C1=3, C2=7

Code Name	***Marks Obt.***	***TA Marks***
1.c	4.0	5
2.c	2.0	2
3.c	3.0	3
4.c	10	10
5.c	3.0	3
6.c	3.0	3
7.c	1.5	2
8.c	8.5	9
9.c	1.5	2
10.c	3.0	3

Table 4.9: Automatic evaluation vs TA's evaluation for Swap.c, C1=3, C2=7

4.2.2 Results

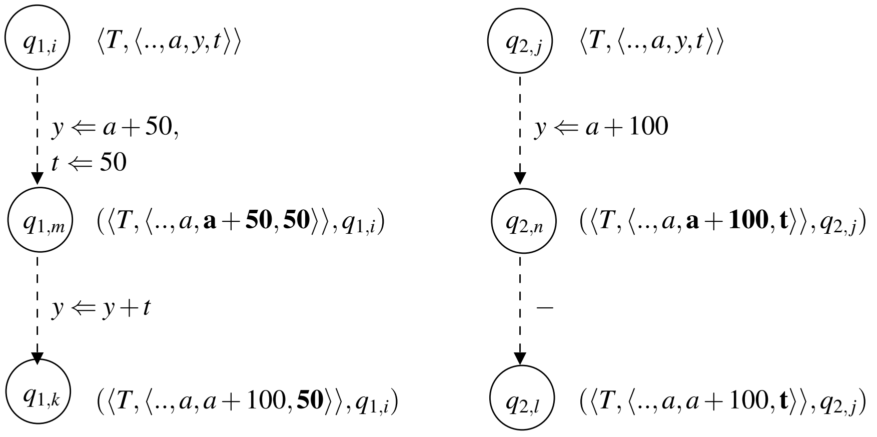
In the tables 4.2 to 4.9, codes were taken from the set of benchmark programs. For example, `GCDBenchmark.c` is a program from the set of benchmark programs. `1.c`, `2.c` etc. are the students' programs.

In the following we explain the concept of value propagation and its use in the marking scheme that is given in the next subsection 4.2.4.

4.2.3 Preliminary concepts of value propagation

The FSMD model, similar to a flowchart, captures the control and data-flow of a program; a detailed description of the model can be found in [47]. Since it is impossible to compare two FSMDs (programs) in totality, they are broken down into smaller segments by introducing cut-points so that each loop in an FSMD is cut in at least one cut-point, thereby permitting any computation of the FSMD to be viewed as a combination of paths. A path α in an FSMD model is a finite alternating sequence of states and connecting edges, starting and ending in cut-points without containing any intermediate cut-point. Two FSMDs M_1 and M_2 are said to be equivalent if for any computation ν_1 (viewed as a combination of paths) in M_1, there is an equivalent computation ν_2 in M_2 and vice versa. Therefore, establishing equivalence at the path-level suffices to determine equivalence of FSMDs. It is worth noting that in this work, the initial state, the final state and the states with more than one outgoing transitions in an FSMD are considered as cut-points. Next, we explain the basic concepts related to value propagation based equivalence checking method for FSMDs as described in [11, 12] which is latter illustrated with an example.

The basic method of value propagation consists in identifying the mismatches in the (symbolic) values of the live variables at the end of two paths taken from two different FSMDs; if mismatches in the values of some live variables are detected, then the variable values (stored as a vector) are propagated through all the subsequent path segments. Repeated propagation of values is carried out until an equivalent path or a path ending in the final state is reached. In the latter case, any prevailing discrepancy in values indicates that the original and the transformed behaviours are not equiva-


$q_{1,i}$ $\langle T, \langle .., a, y, t\rangle\rangle$
$y \Leftarrow a + 50,$
$t \Leftarrow 50$
$q_{1,m}$ $(\langle T, \langle .., a, \mathbf{a+50}, \mathbf{50}\rangle\rangle, q_{1,i})$
$y \Leftarrow y + t$
$q_{1,k}$ $(\langle T, \langle .., a, a+100, \mathbf{50}\rangle\rangle, q_{1,i})$
$q_{2,j}$ $\langle T, \langle .., a, y, t\rangle\rangle$
$y \Leftarrow a + 100$
$q_{2,n}$ $(\langle T, \langle .., a, \mathbf{a+100}, \mathbf{t}\rangle\rangle, q_{2,j})$
$-$
$q_{2,l}$ $(\langle T, \langle .., a, a+100, \mathbf{t}\rangle\rangle, q_{2,j})$

(a) M_1 **(b)** M_2

Figure 4.4: An example of value propagation.

lent; otherwise they are. Propagation of values from a path α_1 to the next path α_2 is accomplished by associating a *propagated vector* at the end state of the path α_1 (or equivalently, the start state of the path α_2). A *propagated vector* $\overline{\vartheta}$ through a path α_1 is an ordered pair of the form $\langle \mathcal{C}, \langle e_1, e_2, \cdots, e_k\rangle\rangle$, where k is the number of distinct variables from both the FSMDs. The first element $\mathcal{C}$ of the pair represents the condition that has to be satisfied at the start state of α_1 to traverse the path and reach its end state with the upgraded propagated vector. The second element, referred to as *value-vector*, consists of e_i, $1 \leq i \leq k$, which represents the symbolic value attained at the end state of α_1 by the variable v_i. To start with, for the initial state, the propagated vector is $\langle \top, \langle v_1, v_2, \cdots, v_k\rangle\rangle$, where $\top$ stands for *true* and $e_i = v_i, 1 \leq i \leq k$, indicates that the variables are yet to be defined. Let $\mathcal{C}_\alpha(\overline{v})$ and $s_\alpha(\overline{v})$ represent respectively the condition of execution and the data transformation of a path α when there is no propagated vector at the start state α^s of α. In the presence of a propagated vector, $\overline{\vartheta_{\alpha^s}} = \langle c_1, \overline{e}\rangle$ say, at α^s, its condition of execution becomes $c_1(\overline{v}) \wedge \mathcal{C}_\alpha(\overline{v})\{\overline{e}/\overline{v}\}$ and the data transformation becomes $s_\alpha(\overline{v})\{\overline{e}/\overline{v}\}$, where $\{\overline{e}/\overline{v}\}$ is called a substitution; the expression $\tau\{\overline{e}/\overline{v}\}$ represents that all the occurrences of each variable $v_j \in \overline{v}$ in τ is replaced by the corresponding expression $e_j \in \overline{e}$ simultaneously with other variables. In the following we present a vivid example for value propagation based checking of equivalence.

Example 4.11. Let us consider the example given in Figure 4.4 where y and a are common variables[2] and t is an uncommon variable and all the states shown in the figure are cut-points. Let the states $q_{1,i}$ and $q_{2,j}$ be corresponding states, i.e., the val-

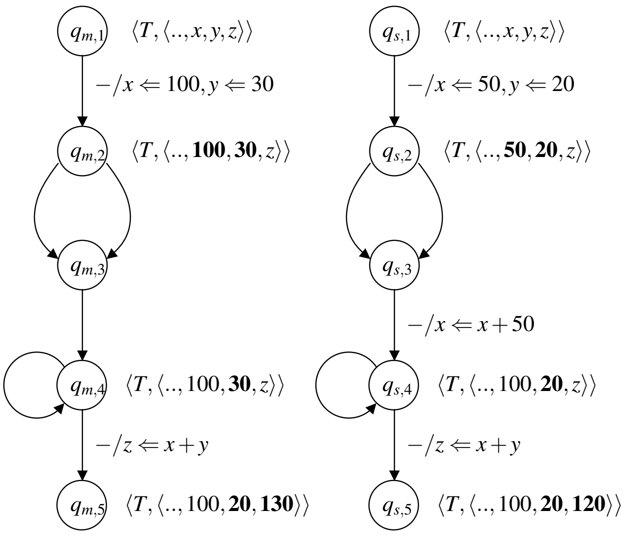
[2] A variable is said to be *common* if it appears in both the FSMDs, whereas it is said to be *uncommon* if it appears in either of the FSMDs but not both.

ues of all the live variables match at these states – consequently, we have identical propagated vectors $\langle T, \langle .., a, y, t \rangle \rangle$ at $q_{1,i}$ and $q_{2,j}$; the ellipsis is used to represent other variables that might be occurring (and whose values also match) in these FSMDs. After the paths $q_{1,i} \twoheadrightarrow q_{1,m}$ in M_1 and $q_{2,j} \twoheadrightarrow q_{2,n}$ in M_2 are traversed, mismatches in the values of the live variables y and t are found and consequently these two paths are considered to be *conditionally* equivalent with the hope that the mismatches would disappear in the respective subsequent path segments from $q_{1,m}$ and $q_{2,n}$. The propagated vectors at $q_{1,m}$ and $q_{2,n}$ are computed accordingly and shown in the figure. With these propagated values, it is easy to check that the values of y (that is, all live variables – assuming t is no further used in FSMD M_1) match at the end of the paths $q_{1,m} \twoheadrightarrow q_{1,k}$ and $q_{2,n} \twoheadrightarrow q_{2,l}$. Therefore, these two paths are declared to be equivalent. ■

We have suggested a symbolic value propagation based scheme below for evaluation of student's programs. In the symbolic scheme, we check live variables at each cut point in the two programs and find equivalence. The value propagation based algorithm for variable mapping has been designed to handle the following situation, if there is an unmapped live variable in path cover. For the two programs, whose variables are to be mapped, the update vectors are checked. If update vectors for both the programs have the same operations, then we can say that both variables will behave in program similarly. For example, let us say $x = y\%10$ and $z = r\%10$ are two statements prior to any given cut points in the two programs. So, x and z are live variables and $\langle y\%10, r\%10 \rangle$ is an update vector. We assume that in the previous path cover y and r have been mapped. Then, x and z will have correspondence in both programs. Hence, the mapping between x and z is true. This is how the variable mapping will work, in a nutshell. Some cases and the basic scheme are explained in detail in next sections.

4.2.4 A value propagation based automated program evaluation scheme

The objective of this work is to evaluate a student's program with respect to a model program (supplied by the instructor). Both of these programs are converted into their corresponding FSMD models and subjected to an automated evaluator which develops upon the FSMD based equivalence checker of [12]. An overview of the scheme is as


$q_{m,1}$
$\langle T, \langle .., x, y, z\rangle\rangle$
$-/x \Leftarrow 100, y \Leftarrow 30$
$q_{m,2}$
$\langle T, \langle .., \mathbf{100}, \mathbf{30}, z\rangle\rangle$
$q_{m,3}$
$q_{m,4}$
$\langle T, \langle .., 100, \mathbf{30}, z\rangle\rangle$
$-/z \Leftarrow x+y$
$q_{m,5}$
$\langle T, \langle .., 100, \mathbf{20}, \mathbf{130}\rangle\rangle$
$q_{s,1}$
$\langle T, \langle .., x, y, z\rangle\rangle$
$-/x \Leftarrow 50, y \Leftarrow 20$
$q_{s,2}$
$\langle T, \langle .., \mathbf{50}, \mathbf{20}, z\rangle\rangle$
$q_{s,3}$
$-/x \Leftarrow x+50$
$q_{s,4}$
$\langle T, \langle .., 100, \mathbf{20}, z\rangle\rangle$
$-/z \Leftarrow x+y$
$q_{s,5}$
$\langle T, \langle .., 100, \mathbf{20}, \mathbf{120}\rangle\rangle$

(a) **(b)**

Figure 4.5: (a) Model FSMD, M_g. (b) Student FSMD, M_s for example 4.12.

follows.

The evaluation scheme consists of two passes:
(i) ALAP (as late as possible) pass: In this pass, variables with mismatches at the final states of two conditionally equivalent paths, one from the model FSMD and other from the student FSMD, are given ample scope to eventually attain identical values at some latter states to account for code motions.
(ii) ASAP (as soon as possible) pass: In this pass, variables in the student FSMD which never attain identical values as those of the model FSMD after the first pass are assigned the same value as that of the model FSMD after the paths containing mismatched definitions are traversed; this is done to determine the difference between the two programs as explained in the following example.

The ASAP and ALAP strategies are based on the observation that a student may have apparently missed out some code in the initial part of his program, which he may have included at a later stage in the program. Hence, a student program should not be penalized for some apparent error in the beginning, which might have been taken care of in the later part of the program.

Example 4.12. Figure 4.5(a) shows the model FSMD and Figure 4.5(b) shows a student FSMD. Our task is to find the difference between the two programs; if no difference is found then the student is awarded full marks, otherwise he is to be marked commensurately based on some metrics.

Pass 1: Initially, when the operations $x \Leftarrow 100, y \Leftarrow 30$ in the model FSMD is compared with the operations $x \Leftarrow 50, y \Leftarrow 20$ in the student FSMD; we find that definitions of both x and y mismatch. However, these mismatches may be later on compensated by some valid code motions applied by the student – so, we optimistically proceed with the respective values in each FSMD. Eventually, when the paths $q_{m,3} \twoheadrightarrow q_{m,4}$ and $q_{s,3} \twoheadrightarrow q_{s,4}$ are compared, we find that the values for the variable x finally match in both the FSMDs. Upon considering the final paths $q_{m,4} \twoheadrightarrow q_{m,5}$ and $q_{s,4} \twoheadrightarrow q_{s,5}$, we find the values of the two variables y and z mismatch.

Pass 2: In this pass, we replace the definitions of the mismatched variables (i.e., variables whose mismatches are retained after the entire first pass) in the student FSMD with those from the model FSMD. Consequently, we replace the operation $y \Leftarrow 20$ by $y \Leftarrow 30$ in the path $q_{s,1} \twoheadrightarrow q_{s,2}$ after this path is compared with $q_{m,1} \twoheadrightarrow q_{m,2}$. The rest of the pass 2 proceeds in a similar fashion as that of pass 1 except when the final paths $q_{m,4} \twoheadrightarrow q_{m,5}$ and $q_{s,4} \twoheadrightarrow q_{s,5}$ are compared; presently, we find that the values of the variable z also match in the two FSMDs. This happens because the difference in the values of y is also reflected when the values of z were computed. Thus, pass 2 helps in eliminating those mismatches which occur as a *ripple effect* of some other earlier mistake(s). ■

Note that we rely on the propagated vectors not only for checking equivalence but also for replacing variable values in the student FSMD during the second pass. These propagated vectors, albeit in a more human-readable form, is also output to the student and the instructor to aid him/her in correcting the programs. While evaluating the student program (FSMD), we keep track of the number of mismatched variables, N_v say, and the number of mismatched paths, N_p say. The marks M given to a student out of the full marks F is calculated based on the following formula:
$M = c_1 \times \frac{T_v - N_v}{T_v} + c_2 \times \frac{T_p - N_p}{T_p}$, where the coefficients c_1 and c_2 are presently supplied by the instructor. T_v and T_p are the total number of variables and paths in the golden program. In addition, the marks obtained can be increased or decreased by another factor which we term as *leniency*, since from our experience we have found that teachers tend to award marks more generously during semester examinations than class tests.

It is to be noted that further automation is possible if the values of c_1 and c_2 are predetermined empirically, thereby reducing the burden on the instructor and increasing fairness in gradation.

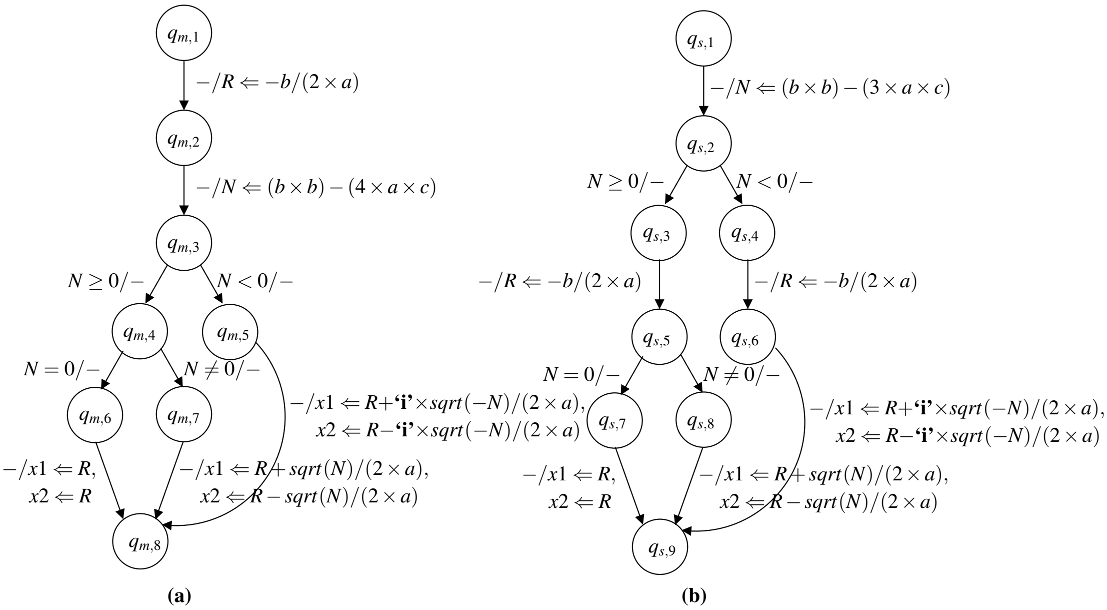

$q_{m,1}$
$-/R \Leftarrow -b/(2\times a)$
$q_{m,2}$
$-/N \Leftarrow (b\times b)-(4\times a\times c)$
$q_{m,3}$
$N \geq 0/-$
$N < 0/-$
$q_{m,4}$
$q_{m,5}$
$N = 0/-$
$N \neq 0/-$
$q_{m,6}$
$q_{m,7}$
$-/x1 \Leftarrow R+$'i'$\times sqrt(-N)/(2\times a)$,
$x2 \Leftarrow R-$'i'$\times sqrt(-N)/(2\times a)$
$-/x1 \Leftarrow R$,
$x2 \Leftarrow R$
$-/x1 \Leftarrow R+sqrt(N)/(2\times a)$,
$x2 \Leftarrow R-sqrt(N)/(2\times a)$
$q_{m,8}$
(a)
$q_{s,1}$
$-/N \Leftarrow (b\times b)-(3\times a\times c)$
$q_{s,2}$
$N \geq 0/-$
$N < 0/-$
$q_{s,3}$
$q_{s,4}$
$-/R \Leftarrow -b/(2\times a)$
$-/R \Leftarrow -b/(2\times a)$
$q_{s,5}$
$q_{s,6}$
$N = 0/-$
$N \neq 0/-$
$q_{s,7}$
$q_{s,8}$
$-/x1 \Leftarrow R+$'i'$\times sqrt(-N)/(2\times a)$,
$x2 \Leftarrow R-$'i'$\times sqrt(-N)/(2\times a)$
$-/x1 \Leftarrow R$,
$x2 \Leftarrow R$
$-/x1 \Leftarrow R+sqrt(N)/(2\times a)$,
$x2 \Leftarrow R-sqrt(N)/(2\times a)$
$q_{s,9}$
(b)

Figure 4.6: An example to illustrate automated program evaluation scheme. (a) Model FSMD, M_g. (b) Student FSMD, M_s.

4.2.5 An illustrative example

In this section, we illustrate the working of our evaluation mechanism outlined in the previous section with the help of a more vivid example.

Example 4.13. Let us consider the assignment where the students have been asked to write a program to find the roots of a quadratic equation using Sridhar Acharya's formula, i.e., if the equation is $a \times x^2 + b \times x + c = 0$, then the two roots for x are given by the formula $\frac{-b \pm \sqrt{b^2 - 4 \times a \times c}}{2 \times a}$. The model FSMD for computing the roots is shown in Figure 4.6(a), the student FSMD is shown in Figure 4.6(b). Note that we have used *sqrt* to represent the function that computes the square root of a number; we have also used the symbol *'i'* to indicate that this *i* is output as a character (to highlight the imaginary part of the root) and has no numerical value associated with it.

Let us begin the equivalence checking procedure.

Pass 1: Initially, the paths $q_{m,1} \twoheadrightarrow q_{m,3}$ and $q_{s,1} \twoheadrightarrow q_{s,2}$ are compared and we find mismatch in the values of the variables R and N. On comparing the paths $q_{m,3} \twoheadrightarrow q_{m,4}$ and $q_{s,2} \twoheadrightarrow q_{s,5}$ we find that the mismatch in R has been resolved, however, mismatch in N persists. A similar situation occurs when the paths $q_{m,3} \twoheadrightarrow q_{m,5}$ and $q_{s,2} \twoheadrightarrow q_{s,6}$ are compared. Finally, when the following pair of paths are compared we find mismatches in three variables namely, N, $x1$ and $x2$: $\langle q_{m,4} \overset{N=0}{\twoheadrightarrow} q_{m,8}, q_{s,5} \overset{N=0}{\twoheadrightarrow} q_{s,9} \rangle$, $\langle q_{m,4} \overset{N \neq 0}{\twoheadrightarrow} q_{m,8},\ q_{s,5} \overset{N \neq 0}{\twoheadrightarrow} q_{s,9} \rangle$ and $\langle q_{m,3} \twoheadrightarrow q_{m,8}, q_{s,2} \twoheadrightarrow q_{s,9} \rangle$.

Pass 2: Initially, the paths $q_{m,1} \twoheadrightarrow q_{m,3}$ and $q_{s,1} \twoheadrightarrow q_{s,2}$ are compared and we again find mismatch in the values of the variables R and N. However, this time we know that the values of N never match, hence we set the value of N to $b \times b - \mathbf{4} \times a \times c$ in the student FSMD. Subsequently, we find a match in the values of all the variables along all the paths. Thus, we conclude that the only mistake that the student has made is in the computation of N in the path $q_{s,1} \twoheadrightarrow q_{s,2}$. ■

As borne out by the above example, the mismatch in the values of R was resolved because we followed an ALAP strategy, while the mismatches for $x1$ and $x2$ got resolved due to our ASAP strategy; the student is finally penalized only for computing N wrongfully which is indeed the only mistake he had made. Considering full marks to be 10, $c_1 = 5$ and $c_2 = 5$, the student is awarded $(5 \times \frac{3}{4} + 5 \times \frac{4}{5}) = 7$ marks.

Discussion

We have observed the following based on the results of our work in this chapter.

1. Approximate equivalence can be established in many cases where normalization does not help, by using randomised simulation similar to as reported in [79].
2. Automated assessment algorithm may be used, as the results for evaluation of programs using the above discussed formula show close resemblances to the manual checking of programs, as is evident from the tables 4.2 to 4.9.

4.3 The benchmark programming assignment suite

In this section, we present the programming assignments designed for evaluation of novice programmers. Benchmark programs [80] have been identified for collecting student programs. Since the assignments should test only the introductory concepts in programming, the solutions to these assignments require preliminary knowledge, such as conditionals, loops, etc., and not of any advanced programming feature. Note that one probable solution for each of the benchmark problems are provided in Appendix A, however, some solutions are neither the most conventional nor most efficient. For example, the lowest common multiple (LCM) of two integers is conventionally computed by dividing the product of the two numbers by their greatest common divisor; however, in the presented solution (*Solution for assignment 8*), we have not taken conventional approach, rather we first store the maximum of the two integers in a variable *lcm* and then keep on incrementing the value stored in the variable *lcm* by one in each step until we reach a value which is divisible by both the integers – this value is output as the LCM of the two integers; such unconventional solutions are often presented by young programmers and hence should be kept in mind during evaluation. On the other hand, the most efficient solution is often not expected from novice programmers and therefore we have not attempted to include most efficient programs in our given solutions. It is important to note that depending on the level of sophistication of an automated evaluator, a representative solution may be required for each of the possible algorithms which may be employed to solve a given problem, i.e., for a given assignment, different model solutions may be provided by the instructor to encompass a wide range of programs which may be submitted by participating students. The benchmark programming assignments are now given below with sample inputs and outputs to help the students; since our primary aim is to aid in the development of automatic evaluators, clearly specifying the format of an expected output actually restricts the range of different outputs which may have to be parsed otherwise by the automated evaluator – imposition of such a restriction is necessary for better evaluation by an automated tool; examples are also provided for some of the assignments for clarification.

Assignment 1. A set of four 2D coordinates (xi, yi), $i = 1,2,3,4$, is given; $(x1, y1)$ is the top left corner and $(x2, y2)$ is the bottom right corner of a rectangle, similarly points with $i = 3,4$ form another rectangle. Find the top left and bottom right corners of the region of intersection of the two given rectangles. If there is no intersection, print -1.

Sample:
Input:
0 4
4 0
2 6
6 2
Output:

2 4
4 2

Assignment 2. Given a number, you have to find the sum of digits in this number and keep on obtaining the sum of digits of the resulting number until you get a single digit; output that single digit.

Example:
Sum of Digits of $12345 = 1+2+3+4+5 = 15$.
Sum of Digits of $15 = 1+5 = 6$.

Sample:
Input:
12345
Output:
6

Assignment 3. *Abundancy* of a number is the ratio of sum of its divisors and the number itself. *Friendly numbers* are those which have same abundancy.

Example:
Abundancy of 6 is $(1+2+3+6)/6 = 2$.
Abundacy of 28 is $(1+2+4+7+14+28)/28 = 2$.
So, 6 and 28 is a friendly number pair. Write a program takes two numbers as input and decides whether they are friendly numbers or not.

Sample:
Input:
6 28
Output:
Yes

Input:
5 25
Output:
No

Assignment 4. Write a program to find out the roots of a quadratic equation: $a.x^2 + b.x + c = 0$, where a, b and c are real numbers; take a, b, c as input and output the roots of the equation.

Sample:
Input:
1 -1 -12
Output:
-3 4

Assignment 5. Given a number as input, output whether it has same number of 1's and 0's in its binary representation.

Sample:
Input:
2
Output:
Yes

Input:
15
Output:
No

Assignment 6. Write a program to check whether an integer taken as input is a power of 2 or not.

Sample:
Input:
32
Output:
Yes

Input:
256
Output:
No

Assignment 7. Write a program to find the greatest common divisor of two integers.

Sample:
Input:
18 30
Output:
6

Input:
75 84
Output:
3

Assignment 8. Write a program to find the lowest common multiple of two integers.

Sample:
Input:
15 21
Output:
105

Input:
14 70
Output:
70

4.3.1 Possible future extensions

There are several areas which are not represented by the above set of benchmark programs, as the suggested benchmarks are for introductory exercises. A next higher level could be introduction of arrays and automatic checking of programs using arrays. The FSMD method of equivalence checking has been improvised for inclusion of arrays in [14] by introducing an extended model namely, finite state machine with data-path *having arrays* (FSMDA). Automated assessment of array based programs by FSMDA method would therefore be possible and a set of benchmark programs can be accordingly suggested.

A domain for further possible extension is object oriented programming (OOP), which is a universally practised paradigm of programming. Some institutions also introduce programming by using some object oriented language, usually C++ or Java. Though they support many common object oriented features, such as data abstraction, classes and objects, etc., there are differences in the way object orientation is supported in one language than it is done in the other, e.g., C++ supports multiple base classes, Java allows single base class. The object oriented features are thus language or implementation specific, hence development of a common suite of benchmark programs poses challenge.

Parallel algorithms and parallelizing code is the emerging trend in programming which is soon going to find widespread use with the availability of multi core and multi-processor CPUs. The programming paradigms, such as MPI, OpenMP and Java threads will become commonplace and might become part of introductory programming. Suggesting benchmark programs for such needs is a challenge for the future, as they are again language specific with widely varying constructs to support parallelization, e.g., while OpenMP uses compiler directives for simple parallel programming, MPI uses library routines for achieving high level of parallelization.

4.3.2 Conclusion

Equivalence checking of expressions, both algebraic as well as transcendental, is not trivial as there cannot be any canonical form of representation for them. In this chapter we proposed that establishing equivalence in such cases can be done by computing their values at various points of interest and checking that the values of equivalent expressions are equal at those points. Simulation can however be done only in the range of interest and that too, at a finite number of points. Equivalence, thus, can not be claimed with 100% confidence.

In this work we chose algebraic as well as trigonometric functions. We will take up exponential, logarithmic functions as well as improve upon the notion of ε-cover in our future work.

Translation validation of programs using FSMD based equivalence checkers has received attention over the years and hence these equivalence checkers can currently handle a plethora of code motions. Leveraging the expertise of the FSMD based equivalence checking method, we aspire to provide a platform for automated evaluation of programming assignments that can account for a wide range of code transformations that has never been targeted before by any automated evaluator. A student program is graded in comparison with a model program supplied by the instructor. Both the programs are converted into corresponding FSMDs before being fed to our system. Our evaluation scheme consists of two passes – an ALAP pass followed by an ASAP pass. The first pass intends to find out all the mistakes made by the student; however, in the process, it does not adjudge a dissimilarity between the student and the model FSMDs as a mistake immediately upon discovery because we cannot rule out the possibility that the student may have taken care of it subsequently in the program. Penalizing the student for all the mismatches that are identified after the first pass may not be a wise choice because a single mistake may cause many mismatches in variable valuations as side effects; the second pass intends to resolve such cases.

Since marking of programming assignments involves human experience which cannot be always quantified as precise mathematical formulas, a lot of empirical studies are required in developing automated program evaluators which we are currently pursuing. A drawback of our method is that it performs well if the control structures of the programs being compared are similar. However, we have found that novice programmers often make such mistakes that may alter the control structure of a program substantially, such as skipping or misplacing parenthesis. Improving our method to attend to such mistakes as well remains as our future goal.

This chapter also presents a suite of benchmark programming assignments to evaluate novice programmers. The assignments and one probable solution for each of these assignments are given in C language. We expect that these benchmarks will contribute towards further advancement of the automated program evaluation community. We have additionally discussed some future challenges involved in extending this suite to address advanced programming courses.

Chapter 5

Conclusion and scope for future work

In this thesis, an attempt has been made to explore the automated assessment of student's programming exercises. This work tends towards categorizing the errors in the submitted program into one of the four possible categories and then tries to identify the errors, correct them and evaluate the program for grading. We conclude the thesis by drawing the attention to the fact that in this work, in most of the places, we have tried to present the proof-of-the concept. The necessary theoretical contributions have been discussed at length, with suitable examples, and the prototypes have been tested using computer programs, which were also developed.

5.1 Summary of contributions

The major contributions of this work are: *i)* extension of equivalence checking method of FSMDs, by checking containment, for the diagnosis of errors in student programs, *ii)* identification of classes of some errors in programs, *iii)* detecting flaws in special constructs such as if-else-if, *iv)* variable mapping for the preprocessing of student programs, *v)* approximate equivalence checking of mathematical expressions, *vi)* suggesting a scheme for automated marking of student programs. The detailed summary of the contributions in this thesis are given in the following sub-sections 5.1.1 - 5.1.3.

5.1.1 Containment analysis of students' programs through equivalence checking

In this part of the work the research objectives were identified as *i)* developing a scheme for containment analysis of students' programs through equivalence checking, *ii)* classifying the errors in programs into broad categories and *iii)* devising strategies for error diagnosis for each class of errors in the programs. Contributions as a result of work done for this part are described below.

The existing path extension based FSMD equivalence checking approach, which compares two given FSMDs having same variable naming and does a depth first analysis for equivalence of paths from one cut point to the next in the two FSMDs, was modified to incorporate a statement containment checking approach. In statement containment checking, it is examined whether the assignment statements in a path are contained in the other FSMD or not. The containment checking mechanism was incorporated along with the equivalence checking and the subsequent experiments for checking the programs were successfully performing containment checking in all cases as expected, thus aiding in the error diagnosis.

Using the above approach of containment checking of the FSMDs based on comparison between cut-point to cut-point path pairs in them, the containment could be divided into following broad categories, viz., *i)* unordered path-wise both way contained, *ii)* path-wise one way contained, and *iii)* path-wise un-contained. These categories of containment can be mapped to various types of errors in the programs with respect to the golden model, which may creep in while programming viz., *i)* dependency violation, *ii)* parenthesis skipping of various types and *iii)* error of missing code segment. The details of these errors and their associated type of containment have been shown in table 2.1. It is evident from this table that in the experiments conducted, containment checking is helpful in identifying the correct error-type.

Algorithms have been developed for reporting the type of error in the student's program and suggesting the correct portion of the code from golden program, corresponding to the erroneous code of student's program. These algorithms have been applied to several test programs and the results have been tabulated in table 2.1. Scheme was proposed for missing nested loop and nested conditions, which also works for code with no loop and conditions as a trivial case. In table 2.2, the schemes for missing nested codes have been shown to be working with the help of simulation examples given in chapter 2 and appendix B.

5.1.2 Methods to reconcile dissimilarities between FSMDs arising from students' programs

In this part of the work preprocessing requirements of programs were aimed at. The research objectives for this part of the work were identified as *i)* developing methods to support automated evaluation, in cases where programs have conditional constructs, which should obey precedences and *ii)* to develop a scheme for variable mapping in programs. Work done for this part is described below.

The first aspect mentioned above is due to the fact that logic in programs demands that in a nesting of if statements, there are conditions that have to be evaluated in a certain order, but the students may violate that order. The student's program, therefore, has to be subjected to a preprocessing step, before equivalence checking is done. The objective of preprocessing is that the nested conditions should conform to some rule of precedence and that a mapping of the names of variables has to be evolved. The complexity and soundness of our method was proved formally. The results of detecting incorrect order of sequences of the conditional construct are summarized in table 3.3. Various sample programs for each of the cases were tested. Each of the sample program was chosen to have wrong sequencing of the conditional construct. With our program we have been able to diagnose wrong ordering in test samples and suggest the correct ordering.

Another contribution in this part of work is the development of algorithms for variable mapping. The path extension based equivalence checking requires that in order to establish the equivalence of two programs, the two programs should have the same variable names. This is a major requirement for checking equivalence, particularly in the automatic assessment of programming assignments, because the student programs will not necessarily have the same naming of variables as given in the model program. Our work in this direction has enabled automated mapping of the corresponding variables in the two programs, which use different variable names and whose equivalence needs to be established.

Since the students will be using variable names different from those in the golden program, we have evolved a method for finding mapping or an association of the variables used in the student's program, with the ones used in the golden model. An algorithm has been developed for variable mapping between two programs. This is done as the FSMD based equivalence checking assumes the variable names to be the same in the two programs under examination, without which the equivalence checking is not possible. The variable mapping algorithm is an FSMD driven algorithm in the sense that it prepares the FSMD models of both the programs, compares their paths for similarity of conditions and data-transformation in a depth first manner and tries to establish a mapping between the variables, which assume equivalent symbolic values after traversing a path from a cut point to the next. The results of variable mapping between various pairs of programs belonging to each of the different cases have been presented in chapter 3, which shows that in most of the cases the results are satisfactory.

5.1.3 Supporting techniques for checking and evaluation of students' programs

The research objectives in this part of the work were to develop supporting techniques for checking and evaluation of students' programs such as *i)* checking equivalence of approximately equivalent expressions and *ii)* develop a marking scheme for the programming exercises.

To address the above research questions, in this work, additional supporting techniques for handling the student programs have been focused. A description of this work is given below.

Checking equivalence of approximately equivalent expressions

Comparison of approximately equivalent functions [78, 79] may be required as an aid to equivalence checking discussed earlier. This may be because equivalence checking is not able to determine such equivalences, where two expressions which are approximately equivalent are to be examined for equivalence. An example of two equivalent expressions could be $sin2\theta$ and $2sin\theta cos\theta$. The equivalence checker cannot find out that these two expressions are equivalent.

In this work, therefore, we have used a randomised simulation based approach in which a function is examined at random points in the domain of the other function. The closeness in their values at most of the points may indicate their approximate equivalence. A randomised decision procedure has been presented to establish the equivalence and the results obtained have established the suitability of the method. The results obtained by running the algorithm 17 for various expressions are given in the table 4.1. From the table it is evident that the method is capable of differentiating between the equivalent and non equivalent expressions.

Marking scheme for the programming exercises

A marking scheme for the programming exercises has been developed and tested to preform well. Based on FSMD equivalence checking, using propagation vectors, we have also suggested an improved method [77]. This method also suggests the ASAP and ALAP marking strategies; they are based on the observation that a student program should not be penalized for some apparent error as it may have been taken care of at a later stage in the program.

An algebraic formula has been suggested to compute marks for the student's program, using some constants, which need to be empirically established. While evaluating the student program (FSMD), we keep track of the number of mismatched variables, N_v say, and the number of mismatched paths, N_p say. The marks M given to a student out of the full marks F is calculated based on the formula. In addition, the marks obtained can be increased or decreased by another factor which we term as *leniency*, since from our experience we have found that teachers tend to award marks more generously during semester examinations than class tests. Automated assessment algorithm may be used, as the results for evaluation of programs using the above discussed formula show close resemblances to the manual checking of programs, as is evident from the tables 4.2 to 4.9.

Next, we summarize the overall flow for automated assessment. In the block diagram of figure 1.9, the sequence of invoking various modules has been shown. The student's program is checked for violation of order of precedences of conditional construct such as if- else if. The program not meeting the precedence is then informed to the student, who can submit his program later, after correcting the precedences. The student's program having the correct precedences and the golden program are then subjected for C to FSMD conversion, thereby generating the respective FSMDs. The two FSMDs thus obtained cannot be subjected to equivalence checking just at this stage, because the student's program may use a different set of variable names than the golden program. The variables in the FSMD of student's program have therefore to be renamed first and then we can have the equivalence checked. The current implementation supports the variable renaming as a separate module, however, it can be modified to do variable renaming and equivalence checking hand in hand, as it proceeds with the equivalence checking, starting from the start states of both the FSMDs. In the current implementation, however, equivalence checking is to be followed by variable renaming. In equivalence checking, if it is found that the two FSMDs are equivalent, then the student's program is correct, however if such is not the case then the containment analysis is done to find out how much of the correct code has been written by the student. As a result of first time containment checking, the student's FSMDs is declared to have one of the four types of containment as compared with the golden program, thus indicating some error, which is informed to the student for correction. After first correction, the student may submit again for subsequent equivalence checking, and to find out the next error. This loop may be continued till the student's program is free from all errors. The loop may be mechanized in the future work. The current implementation assumes the student to be in the loop, doing the correction and resubmitting, if needed. The evaluation then can be done on the basis on the number of times the errors were have to be corrected. Presently we rely on a path equivalence and variable mapping based evaluation of the programs, which has been found to give satisfactory results, as discussed in chapter 4.

5.2 Future work

Elaborate implementation of all the concepts in the form of a software, thus developing a unified framework has not been attempted, which is a direction for our work in the future.

Another direction of work for the future is the enhancement of the FSMD model for inclusion of pointers, structures, functions etc. In future we intend to incorporate more errors and test our method further with an enhanced set of test programs and real life programs as well.

5.2.1 Enhancement of FSMD for I/O statements, Pointers

In Chapters 2 and 3, we have presented a path-extension based method and a symbolic value propagation based method for the FSMD model for different purposes. A significant deficiency of this method is its inability to handle the other important classes of programs, namely those involving pointers, functions, structures and arrays. In this chapter we enhance the formal model of FSMD for incorporating the aforesaid features. We enhance the grammar for inclusion of necessary normalization and thus suggest the enhancements in the data structure to include new features of pointers, functions, structures and arrays. We also propose the enhancement in the equivalence checking and for these features and prove the correctness of such enhancements.

The data flow analysis for programs using pointers is notably more complex than those involving no indirection. To illustrate the fact, let us consider two statements, the first from a student's program written as $c = *a + *b$ and the other one in the model program, written as $*c = *a + *b$. Clearly in the first statement, c is not a pointer variable, whereas this is not the case in the model program. We first address the problem of deriving a succinct representation of expressions involving pointers so that the computation of the conditions and data transformations of paths can avoid case analysis.

The rest of this discussion is as follows:

Pointers

Pointers are inevitable components of C program. In this section we first describe the modification in the normalization grammar for inclusion of pointers. A pointer is a special kind of a variable. we have modified the grammar for inclusion of pointers as follows:

Traditionally the normalization technique described in [75] has been used to represent arithmetic and logical expressions e.g. in [11, 14, 44, 47, 54]. The equivalence determination problem of two arbitrary arithmetic expressions over integers is undecidable. Normalization of arithmetic expressions targets the structural similarity of two such expressions as the first step; in the process, many equivalent formulae become syntactically identical.

It is possible to convert any arithmetic expression, on application of the grammar rules in [75], involving integer variables and constants into its normalized form. The set of grammar rules has been revised by updating the rule (3) and incorporating the new rules (6) and (7), as given below, to accommodate pointers.

Extended grammar:

1. $S \rightarrow S+T\big|c_s$, where c_s is an integer.
2. $T \rightarrow T*P\big|c_t$, where c_t is an integer.
3. $P \rightarrow \text{abs}(S)\big|(S)\,\text{mod}(C_d)\big|f(listS)\big|S \div C_d\big|v\big|(*(S_p))\big|c_m$, where v is a variable, and c_m is an integer.
4. $C_d \rightarrow S \div C_d\big|(S)\,\text{mod}(C_d)\big|S$,
5. $list\ S \rightarrow list\ S,\ S\big|S$.

   the following rules have been added to separate the pointer arithmetic from the integer arithmetic in the expression for a normalized sum.

6. $S_p \rightarrow D_p + 1*(\&v)$
7. $D_p \rightarrow 0 + c_s * I_{sz}\big|0$, where c_s is a +ve integer constant and I_{sz} is the size of integer pointer

In the above grammar, the non-terminals S, T, P stand for (normalized) sums, terms and primaries, respectively, A is an array primary, and C_d is a divisor primary. The nonterminal S_p stands for normalized sum in an expression involving pointers. S_p produces the normalized expression for the index of a variable or that of an indexed location (latter one is for future use, in case arrays are incorporated). The terminals are the variables belonging to $I \bigcup V$, the interpreted function constants abs, mod and $\div$ and the user defined uninterpreted function constants f. In addition to the syntactic structure, all expressions are ordered as follows: any normalized sum is arranged by lexicographic ordering of its constituent subexpressions from the bottom-most level, i.e., from the level of simple primaries assuming an ordering over the set of variables

$I \bigcup V$; among the function terminals, abs $\prec \div \prec$ mod $\prec$ uninterpreted function constants. As such, all function primaries, including those involving the uninterpreted ones, are ordered in a term in an ascending order of their arities.

An example of use of above grammar is as follows

Example 5.1. Consider the following lines of code

```
int y, *x;
........
y = *x + y;
```

the rhs expression of this assignment operator will appear as `0+1*(*x)+1*y` - note that the `*` occurring within the parenthesis with `x` is a primary in the normalized sum corresponding to the rhs expression.

Explanation of rule 6:
An example of use of rule 6 is as follows

Example 5.2. In the following code

```
int v, *x;
x = &v;
```

the rhs "`&v`" will appear as the normalized sum "`(0+1*(&v))`". Note that "`(0+1*(&v))`" will be parsed as "S_p" using rule 6, with D_P as 0, as shown in figure 5.1.

Input and output statement representation in NCell

While finding the equivalent path for a path, it is required to check the equivalence of the respective conditions as well as the data transformations of the paths. Since the condition of execution and the data transformation of a path involve the whole of integer arithmetic, checking equivalence of paths reduces to the validity problem of first order logic which is undecidable; thus, a canonical form does not exist for integer arithmetic.

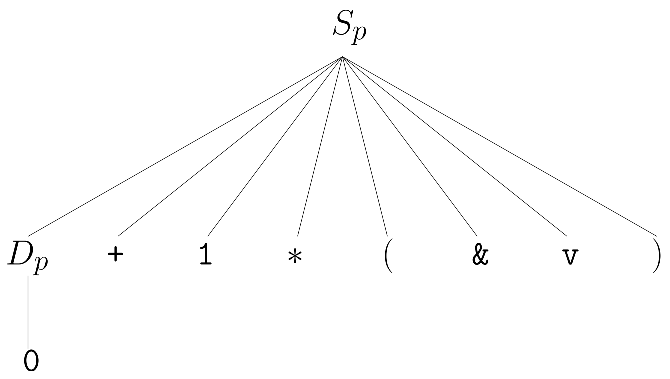


Figure 5.1: Parse tree for &v

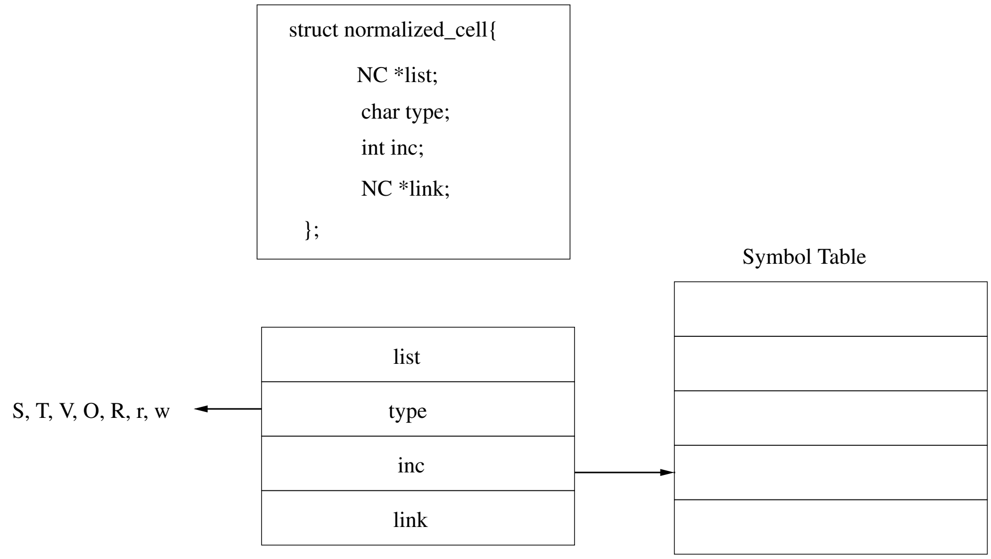
The normalization process reduces many computationally equivalent formulas syntactically identical as it forces all the formulas to follow a uniform structure. In the following, the normal form chosen for the formulas and the simplification carried out on the normal form during the normalization phase are briefly described.

A condition of execution (formula) of a path is a conjunction of relational and Boolean literals. A Boolean literal is a Boolean variable or its negation. A relational literal is an arithmetic relation of the form s R 0, where s is a normalized sum and R belongs to <=,>=,==,. The relation >(<) can be reduced to >=(<=) over integers. For example, $x-y>0$ can be reduced to $x-$ ($y-1$) $>=0$.

The data transformation of a path is an ordered tuple $\langle e_i \rangle$ of algebraic expressions such that the expression e_i represents the value of the variable v_i after execution of the path in terms of the initial data state. So, each arithmetic expression in data transformation can be represented in the Normalized Sum form. A normalized sum is a sum of terms with at least one constant term; each term is a product of primaries with a non-zero constant primary; each primary is a storage variable, an input variable or of the form $abs(s)$, $mod(s_1,s_2)$, $exp(s_1,s_2)$ or $div(s_1,s_2)$, where s, s_1, s_2 are normalized sums. These syntactic entities are defined by means of production of the following grammar.

Grammar of the normalized sum

1. $S \rightarrow S+T \mid c_s$, where c_s is an integer.



```
struct normalized_cell{
        NC *list;
        char type;
        int inc;
        NC *link;
  };
```

Figure 5.2: Data Structure used for NCell

2. $T \rightarrow T * P \mid c_s$,where c_s is an integer

3. $P \rightarrow abs(S) \mid (S)mod(S) \mid S/C_d \mid v \mid c_m$, where v and c_m are an integer.

4. $C_d \rightarrow S/C_d \mid S$.

Thus, the exponentiation and the (integer) division are depicted by infix notation and all functions have arguments in the form of normalized sums. In addition to the above structure, any normalized sum is arranged by lexicographic ordering of its constituent sub expressions from the bottom-most level, i.e., from the level of simple primaries.

Data structure used for NCell The data structure to implement an NCell can be viewed in figure 5.2.

Representation of the normalized expressions

All normalized expressions are represented by NCell, the tree structure, which is implemented by linked lists [75]. Each node in the tree is a normalized cell consisting of the following four fields :

- A **LIST**-pointer , which points to the entries at the same level of the tree or equivalently, at the same hierarchical level of an expression.

- A **TYPE**-field which indicates the type of the cell. Some typical examples of the types are ‘S’ for normalized sum, ‘T’ for normalized term, ‘R’ for relational literal, etc. TYPE = ‘v’ indicates a program variable or more generally a symbolic constant.

- An integer field **INC** , the meaning of which varies from type to type. For example, TYPE = ‘S’, INC = 4 means that the integer constant in the normalized sum is 4.

- A **LINK**-pointer , which points to the leftmost successor of the node in question in the next level of the tree or equivalently, in the next syntactic level of the expression.

The difference between the LIST-pointer and the LINK-pointer is noteworthy. For example, the non-constant terms of a sum are connected by LINK-ing the first term to a normalized cell of TYPE ‘S’ and LIST-ing the other terms starting from the first term onwards.

It may be noted that the equivalence checker works on the expressions stored in NCells.

Grammar used to represent I/O statement in NCell The modified grammar that represent an I/O statement in FSMD using NCell is given as follows:

1. $S \rightarrow S+T \mid c_s \mid r \mid w$, where c_s is an integer, r is used for read / scanf statement and w is used for write / printf statement.
2. $T \rightarrow T*P \mid c_s$,where c_s is an integer
3. $P \rightarrow abs(S) \mid (S)mod(S) \mid S/C_d \mid v \mid c_m$, where v and c_m are an integer.
4. $C_d \rightarrow S/C_d \mid S$.

Data transition for read write statement can be viewed in FSMD as in figure 5.3.

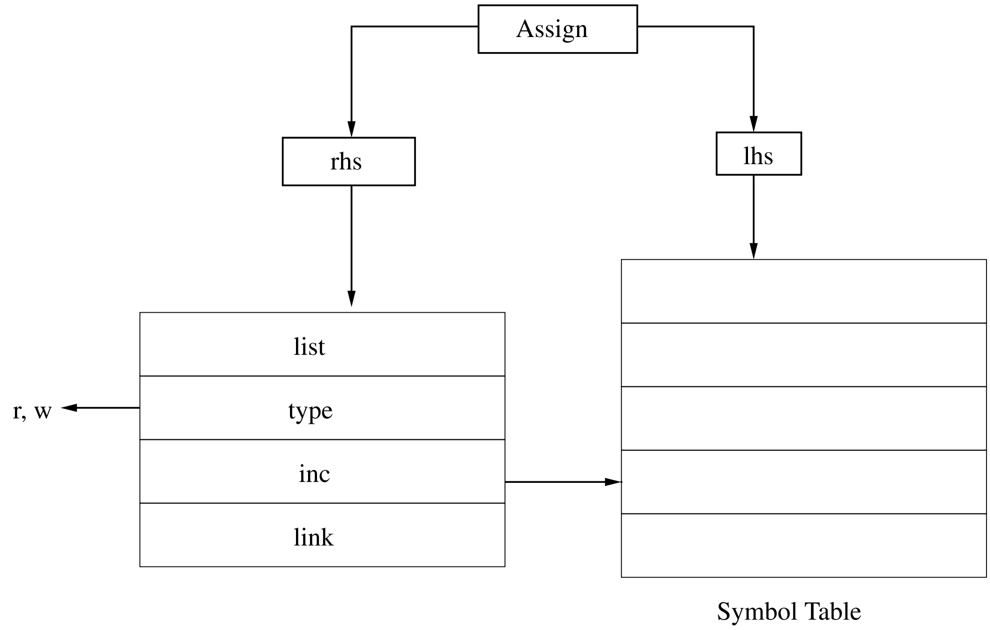


Figure 5.3: Data Structure used for read/ write

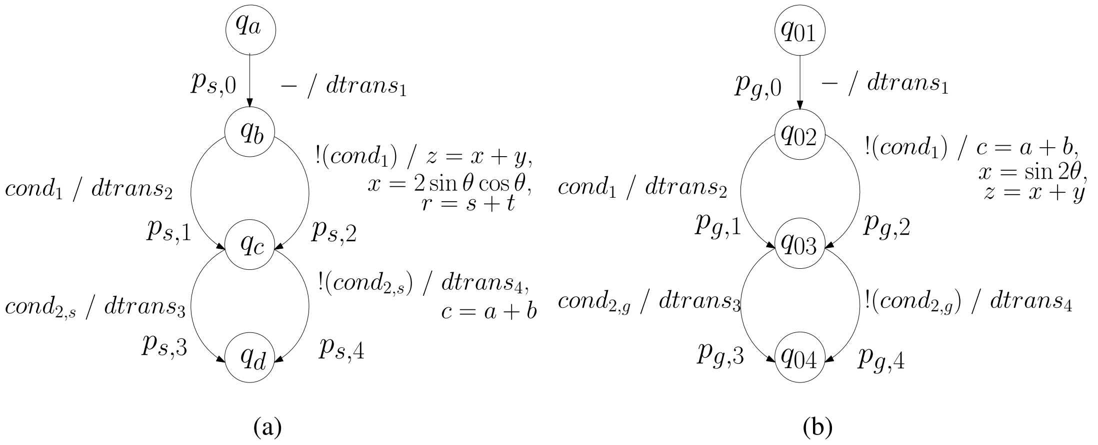
5.2.2 Future work in variable mapping

Presently variable mapping is done as a pre-processing step. This may be enhanced in the future to work hand-in-hand with the equivalence checking and containment checking steps.

5.2.3 Future work in approximate equivalence checking: Extension of containment checking by both ways path extension for approximate equivalence checking

In the following we present an example of checking and correcting the error in a student program using approximate equivalence checking of the FSMDs of student's program and the golden program. Here we use an improved version of containment checker, capable of extending paths in both the FSMDs, in order to establish containment of extended paths.

Example 5.3. The FSMDs of the student's program and the golden program are shown in figure 5.4. We note that the equivalence checker finds two paths computationally equivalent, if the conditions and the data-transformations on both match. We also note that in containment checking, a path p_s is said to be contained in another, say p_g, if the data transformations in p_s are also present in p_g, irrespective of their conditions. The checking and correction process is executed in the following steps.


q_a
$p_{s,0}$
$-$ / $dtrans_1$
q_b
$!(cond_1)$ / $z = x + y$, $x = 2\sin\theta\cos\theta$, $r = s + t$
$cond_1$ / $dtrans_2$
$p_{s,1}$
q_c
$p_{s,2}$
$!(cond_{2,s})$ / $dtrans_4$, $c = a + b$
$cond_{2,s}$ / $dtrans_3$
$p_{s,3}$
q_d
$p_{s,4}$
q_{01}
$p_{g,0}$
$-$ / $dtrans_1$
q_{02}
$!(cond_1)$ / $c = a + b$, $x = \sin 2\theta$, $z = x + y$
$cond_1$ / $dtrans_2$
$p_{g,1}$
q_{03}
$p_{g,2}$
$cond_{2,g}$ / $dtrans_3$
$!(cond_{2,g})$ / $dtrans_4$
$p_{g,3}$
q_{04}
$p_{g,4}$

(a) (b)

Figure 5.4: FSMDs (a) M_s and (b) M_g.

Equivalence checker is executed in the beginning. It finds the path $q_a \rightarrow q_b$ (path $p_{s,0}$ in M_s), equivalent to the path $q_{01} \rightarrow q_{02}$ (path $p_{g,0}$ in M_g), as they have the same condition (-) and data-transformation ($dtrans_1$). q_b and q_{02} become corresponding states.

Equivalence checker then finds the equivalence of the paths $q_b \rightarrow q_c$ (path $p_{s,1}$ in M_s), equivalent to the path $q_{02} \rightarrow q_{03}$ (path $p_{g,1}$ in M_g), as they have the same condition ($cond_1$) and data-transformation ($dtrans_2$).

After this, the equivalence checker fails to find the equivalence of the path $q_b \rightarrow q_c$ (path $p_{s,2}$ in M_s), with the path $q_{02} \rightarrow q_{03}$ (path $p_{g,2}$ in M_g), as they have the same condition ($!(cond_1)$) but not the same set of data-transformations along the two paths.

As a result, q_b becomes LCS and q_{02} becomes CSLCS, from where the containment checker starts checking. It finds $p_{s,1}$ contained in $p_{g,1}$, as the data-transformation along them are equal ($dtrans_2$).

The containment of $p_{s,2}$ is not found in $p_{g,2}$, even after extending the path $p_{g,2}$ further, as the set of data-transformations in $p_{s,2}$ is not a subset of the set of data-transformations in $p_{g,2}$. The dead code $r = s + t$ is making the containment check fail, so it once it is marked and ignored by the containment checker, the containment of $p_{s,2}$ is established in $p_{g,2}$. Also the statement $c = a + b$ is not copied to $p_{s,2}$ now as it may occur at a later stage. As the equivalence checker failed but containment checker succeeded, so the two dependent data-transformations are interchanged in the $p_{s,2}$, making the two paths equivalent, which will be seen in the next step.

Again, equivalence checker is executed to check the extent of equivalence of the two FSMDs. This time equivalence checker is able to find equivalence of both the paths $p_{s,1}$ and $p_{s,2}$ emanating from q_b and meeting at q_c with the paths $p_{g,1}$ and $p_{g,2}$ emanating from q_{01} and meeting at q_{02}. Thus q_c and q_{02} become corresponding states.

Equivalence checker starts to explore the paths emanating from the recently established corresponding states q_c and q_{02}, but it fails to check further due to mismatch in the conditions. Thus q_c becomes new LCS and q_{02} becomes new CSLCS.

The containment checker is now called and it starts finding containment of path $p_{s,3}$ from LCS to q_d in the path $p_{g,3}$ from CSLCS to q_{03}. The data-transformation being the same along the two paths, containment is asserted. As the conditions are different, so the condition of the path $p_{s,3}$ is replaced with the condition of the path $p_{g,3}$ of golden program.

The containment checker now checks containment of path $p_{s,4}$ from LCS to q_d in the path $p_{g,4}$ from CSLCS to q_{03}. The data-transformation being the same along the two paths, containment is asserted, ignoring and removing the dead code $c = a + b$ over the path path $p_{s,4}$. As the conditions are different, so the condition of the path $p_{s,4}$ is replaced with the condition of the path $p_{g,4}$ of golden program.

Finally the equivalence checker is executed to establish that both the FSMDs are equivalent.

5.2.4 Future work in FSMD to C conversion

Present work does not use an FSMD to C converter. The tools should have the capability to construct the C-code from a given FSMD. This is an important aspect having many uses. This may be taken up as a future work.

5.2.5 Future work in debugging of evolving programs

An interesting study would be to check the applicability of the methods developed in this thesis to establish equivalence of evolving programs [66].

5.2.6 Future work in debugging programs involving bit-vectors

A future work may be to check programs involving bit-vectors, e.g., the two equivalent versions of shift-add multiplication algorithm coded in Verilog. Work done by our research group to solve this problem [71], which adopts the notions of path covers and symbolic value propagation; however, there remain many challenges still to explore.

Appendix A

Golden solution for each of the programming assignments

In this section, we provide one golden solution for each of the programming assignment given in Section 4.3 in C language. It is to be noted that obtaining the solutions in other high-level languages, such as Java, is easy and straightforward from the programs given here, since the solutions involve preliminary programming constructs that appear, almost identically, in other languages as well.

Solution for assignment 1

```
void main()
{
  int x1, x2, x3, x4, y1, y2, y3, y4;
  int left, right, top, bottom;
  printf("\nEnter x1 and y1:");
  scanf("%d%d",&x1,&y1);
  printf("\nEnter x2 and y2:");
  scanf("%d%d",&x2,&y2);
  printf("\nEnter x3 and y3:");
  scanf("%d%d",&x3,&y3);
  printf("\nEnter x4 and y4:");
  scanf("%d%d",&x4,&y4);
  if( (x1 > x4) || (x3 > x2) ||
      (y1 < y4) || (y3 < y2) )
    printf("-1\n");
  else
  {
    if(x1 > x3)
```

```
      left = x1;
    else
      left = x3;
    if(y1 < y3)
      top = y1;
    else
      top = y3;
    if(x2 < x4)
      right = x2;
    else
      right = x4;
    if(y2 > y4)
      bottom = y2;
    else
      bottom = y4;
    printf("%d %d\n",left,top);
    printf("%d %d\n",right,bottom);
  }
}
```

Solution for assignment 2

```
void main()
{
  unsigned int n, sum;
  printf("\nEnter number:");
  scanf("%d",&n);
  while(n > 9)
  {
    sum = 0;
    while(n > 0)
    {
      sum = sum + n%10;
      n = n/10;
    }
    n = sum;
  }
  printf("%d\n",n);
}
```

Solution for assignment 3

```
void main()
{
  int x, y, sumx, sumy, i;
  double abunx, abuny;
  printf("\nEnter two numbers:");
  scanf("%d%d",&x,&y);
  sumx = 0;
  for(i = 1; i <= x; i++)
  {
    if((x%i) == 0)
      sumx = sumx + i;
  }
  sumy = 0;
  for(i = 1; i <= y; i++)
  {
    if((y%i) == 0)
      sumy = sumy + i;
  }
  abunx = ((double) sumx) / x;
  abuny = ((double) sumy) / y;
  if(abunx == abuny)
    printf("Yes\n");
  else
    printf("No\n");
}
```

Solution for assignment 4

```
void main()
{
  double a, b, c, d, root1, root2;
  printf("\nEnter a, b and c where
          a*x*x + b*x + c = 0:");
  scanf("%d%d%d",&a,&b,&c);
  d = b*b - 4*a*c;
  if (d < 0)
  { //complex roots
    printf("First root = %.2lf + j%.2lf\n",
            -b/(double)(2*a), sqrt(-d)/(2*a));
    printf("Second root = %.2lf - j%.2lf\n",
            -b/(double)(2*a), sqrt(-d)/(2*a));
  }
  else
  { //real roots
    root1 = (-b + sqrt(d))/(2*a);
    root2 = (-b - sqrt(d))/(2*a);
    printf("First root = %.2lf\n", root1);
    printf("Second root = %.2lf\n", root2);
  }
}
```

Solution for assignment 5

```
void main()
{
  unsigned int n, r, count0, count1;
  printf("\nEnter a number:");
  scanf("%d",&n);
  count0 = 0;
  count1 = 0;
  while(n > 0)
  {
    r = n % 2;
    if(r == 0)
      count0 = count0 + 1;
    else
      count1 = count1 + 1;
    n = n / 2;
  }
  if(count0 == count1)
    printf("Yes\n");
  else
    printf("No\n");
}
```

Solution for assignment 6

```
void main()
{
  int n, flag = 0;
  printf("\nEnter a number:");
  scanf("%d",&n);
  while(n != 1)
  {
    if(n%2 != 0)
    {
      flag=1;
      break;
    }
    n = n / 2;
  }
  if(flag == 0)
    printf("Yes\n");
  else
    printf("No\n");
}
```

Solution for assignment 7

```
void main()
{
  int y1, y2, res, i;
  printf("\nEnter two numbers:");
  scanf("%d%d",&y1,&y2);
  res = 1;
  while(y1 != y2)
  {
    if(y1%2 == 0)
    {
      if(y2%2 == 0)
      {
        res = res * 2;
        y1 = y1 / 2;
        y2 = y2 / 2;
      }
      else
        y1 = y1 / 2;
    }
    else if(y2%2 == 0)
      y2 = y2 / 2;
    else if(y1>y2)
      y1 = y1 - y2;
    else
      y2 = y2 - y1;
  }
  res = res * y1;
  printf("%d\n",res);
}
```

Solution for assignment 8

```
void main()
{
  int x,y,lcm;
  printf("\nEnter two integers:");
  scanf("%d%d",&x,&y);
  if(x>y)
    lcm=x;
  else
    lcm=y;
  while((lcm%x)!=0 || (lcm%y)!=0)
  {
    lcm++;
  }
  printf("%d\n",lcm);
}
```

Table A.1: Table for comparison of various features of benchmark programs

Problem no.	Benchmark	No of statements	# if-else	# loops	# maximum nesting	# assignment statements	# basic blocks	# input statements	# output statements
1	# Rectangle intersections	23	5	0	0	8	12	8	4
2	Sum of digits	9	0	1	2	4	5	1	1
3	Friendly numbers	18	3	2	1	6	8	2	1
4	Quadratic roots	12	1	0	0	3	3	3	2
5	Equal 1s and 0s	12	2	1	1	6	7	1	1

Appendix B

Examples for handling cases of missing code of nested loops

As described in Chapter 2, further examples for handling the cases of missing code of nested loops are given below in this appendix.

Example B.1. The FSMDs and codes of golden and student's programs for calculating the sum of digits of the sum obtained until we get a single digit number are reproduced below (Figures B.1 and B.2). The program for this problem requires two loops. In student program the outer while loop is missing, making the student program inconsistent with teacher's program. We discuss below the simulation of the mechanism.

digitsum_correct.c

```
#include<stdio.h>
int main() {
        int n, i, sum, t;
        printf("enter the number");
        scanf("%d", &n);
        sum = n;
        while (sum > 9) {
                n = sum;
                sum = 0;
                while (n > 9) {
                       t =n % 10;
                       sum = sum + t;
                       n = n / 10;
                }
                sum = sum + n;
```

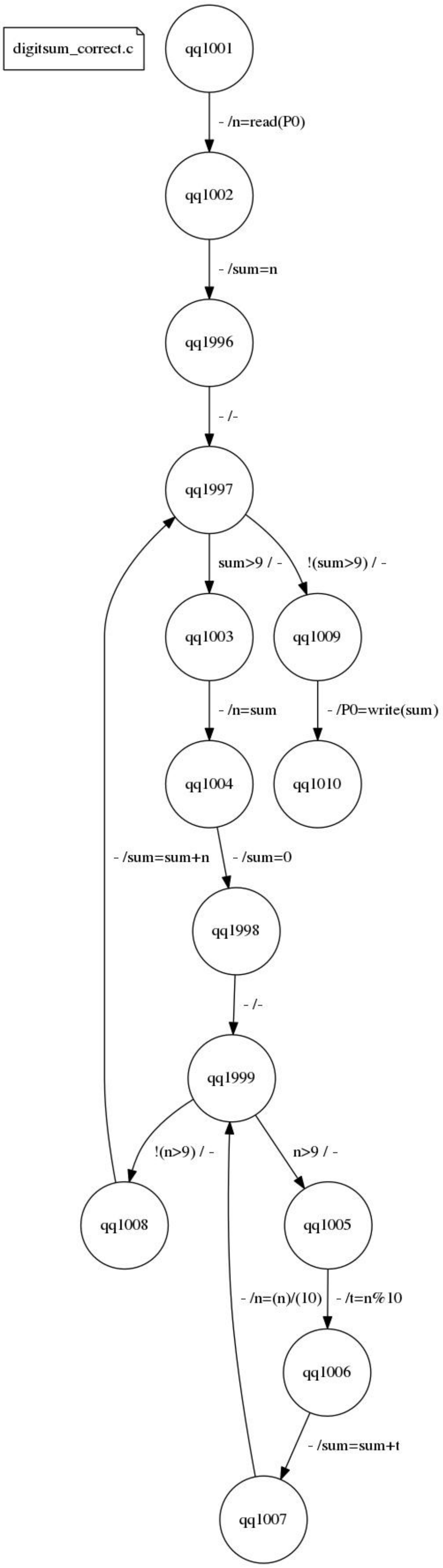


Figure B.1: M_g, digitsum program

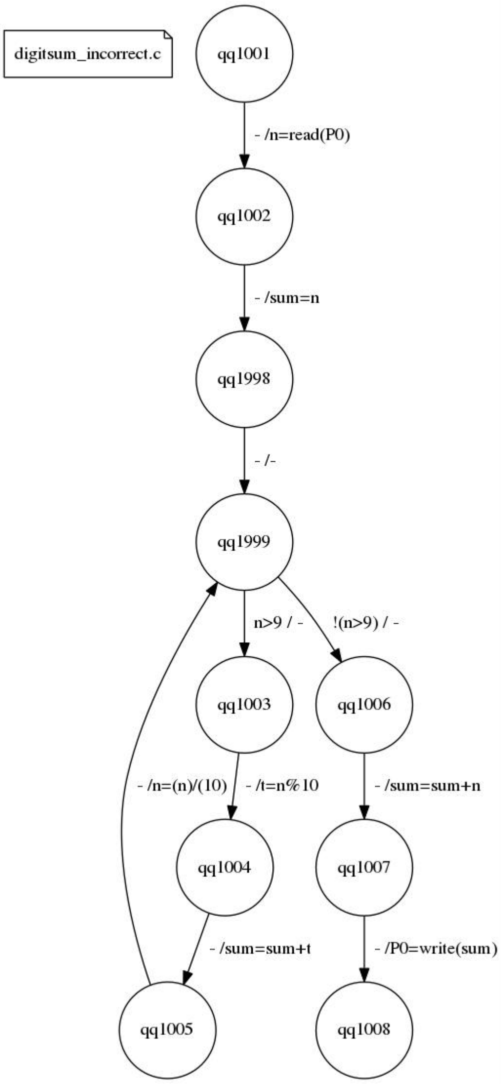


Figure B.2: M_s, digitsum program

```
        }
        printf("%d", sum);
        return 0;
}
```

digitsum_incorrect.c

```
#include<stdio.h>
int main() {
        int n, i, sum, t;
        printf("enter the number");
        scanf("%d", &n);
        sum = n;
        while (n > 9) {
             t =n % 10;
             sum = sum + t;
             n = n / 10;
        }
        sum = sum + n;
        printf("%d", sum);
        return 0;
}
```

Execution of equivalence checker on these two FSMDs gives us $qq1999$ as the LCS. The CSLCS is the state $qq1997$. Containment checker takes the path p_s = $qq1999 \rightarrow qq1003 \rightarrow qq1004 \rightarrow qq1005 \rightarrow qq1999$, starting from $qq1999$ to next cut point i.e $qq1999$ itself in M_s. It tries to find whether this path is present in M_g or not. Containment checker outputs the path $p_g = qq1997 \rightarrow qq1003 \rightarrow qq1004 \rightarrow qq1998 \rightarrow qq1999 \rightarrow qq1005 \rightarrow qq1006 \rightarrow qq1007 \rightarrow qq1999$ of M_g, which is containing the path p_s in M_s. After the execution of step (3), we get the state $qq1999$ in M_s as US. The corresponding state in M_g, the state $qq1997$ is thus CSUS. We associate the variable state names s_g and s_s with CSUS and US respectively. DFS visit of M_g with s_g i.e.,$qq1997$ as root gives that $qq1997$ is the starting of a loop. In step (3) it introduces a self-loop, loop1, in M_s, corresponding to the loop at state $qq1997$ of M_g. This self-loop is introduced at a new state $qq1004$ in M_s, just before $qq1999$, the US (i.e., s_s). The condition of self-loop is kept the same as that of s_g. Now s_g moves down to the next state i.e., $qq1003$ in M_g. The s_s remains at the US.

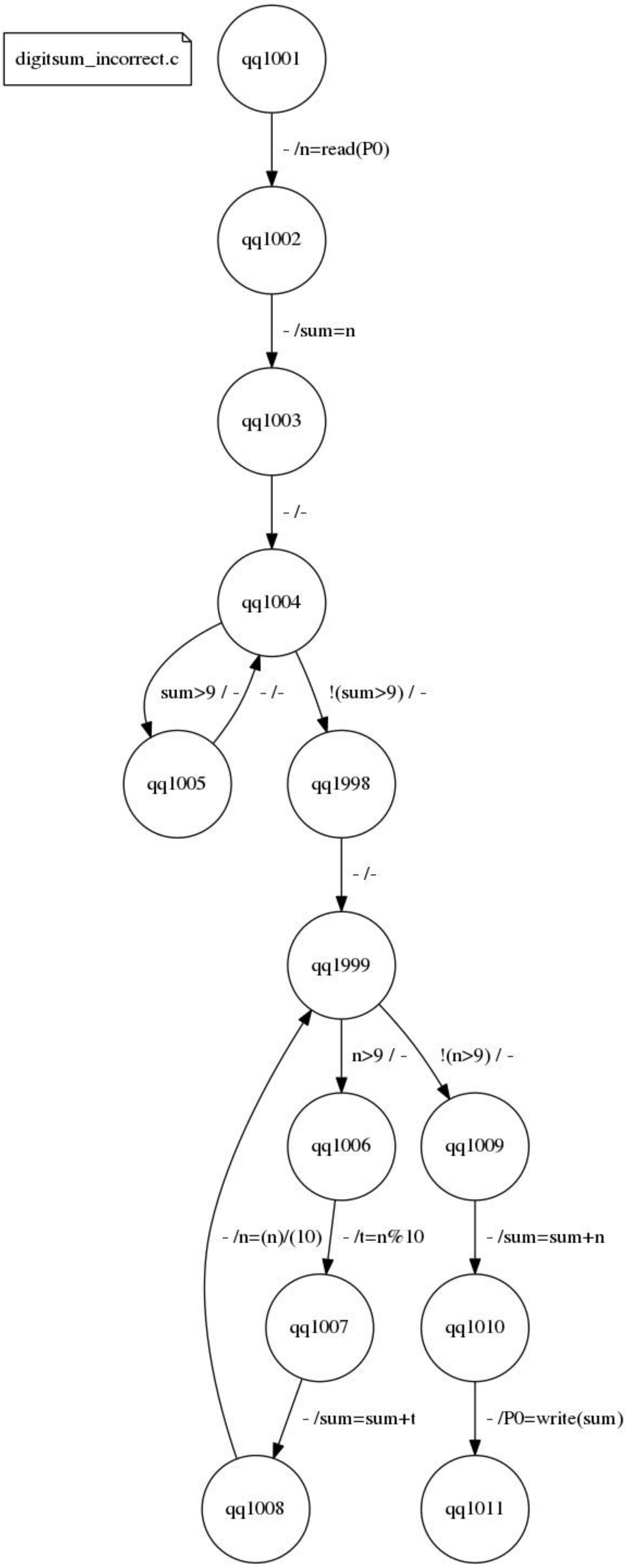


Figure B.3: Modified incorrect FSMD after introducing loop1 before the unmatched state in M_s for digitsum program.

Now s_g moves down to the next state i.e., $qq1003$. The s_s remains at the US, this indicates that the FSMDs are equivalent upto s_g and s_s. As s_g is neither a loop nor a cut-point, so a chain, ChainMg, is made upto the next cut-point i.e. $qq1999$. A chain, ChainMs is made from s_s to the next cut-point, which is also s_s. The chain copy mechanism copies the chain in the loop1. The modified loop1 is shown in B.4.

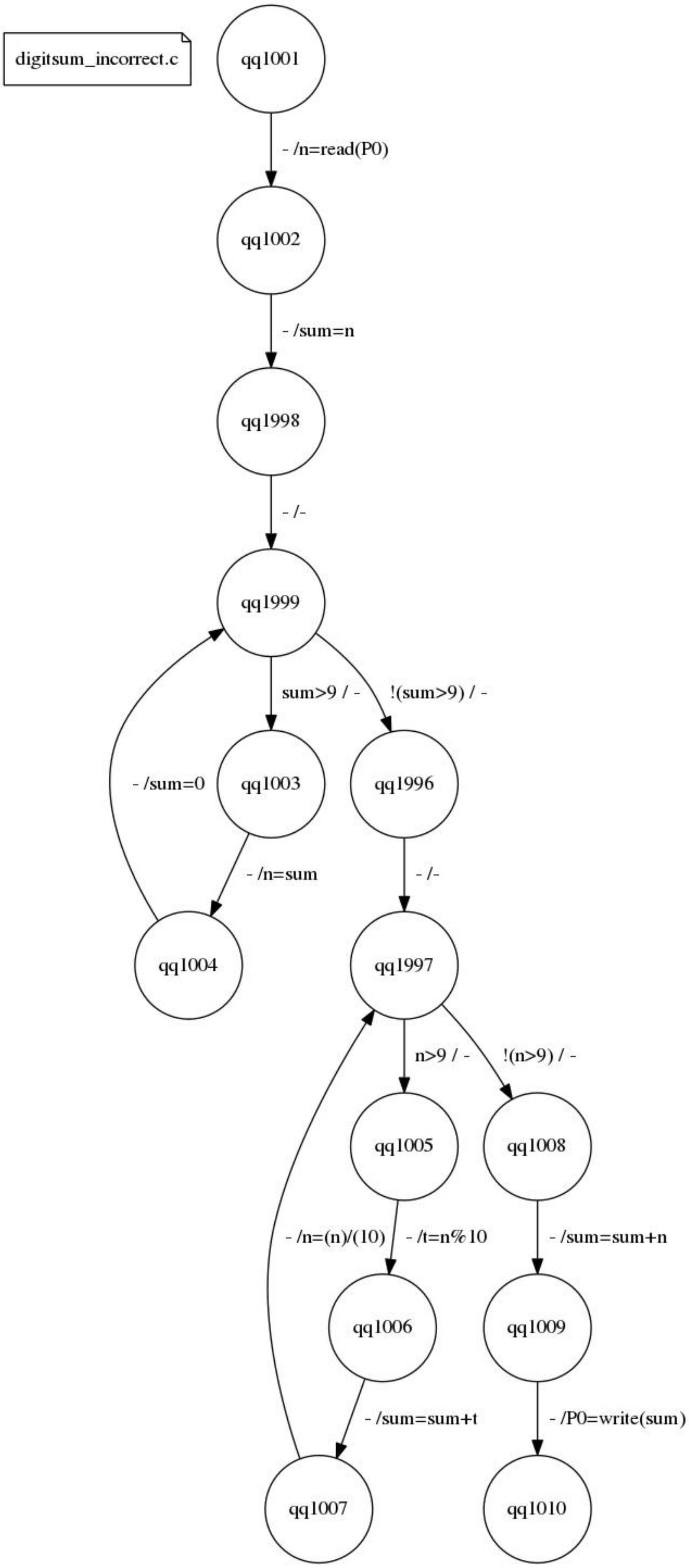


Figure B.4: Incorrect FSMD after modifying loop1 in M_s for digitsum program.

The s_s is still at US. Now the s_g moves down to the state $qq1999$, which is a loop start state. A self-loop, loop2, is therefore introduced inside loop1. The loop condition of loop2 is made the same as the condition of s_g. The conditions of s_g and s_s are same, so both of them move down. s_g further moves down in M_g to the state $qq1005$. Also s_s now moves down to the next state in M_s. As the next states of M_g and M_s are neither cut-point nor loop start states, so the chain copy mechanism prepares the chain from next state of s_g to the next cut-point (which happens to be s_g again). This chain is copied to the loop2. The FSMD of modified student's program is shown in figure B.5.

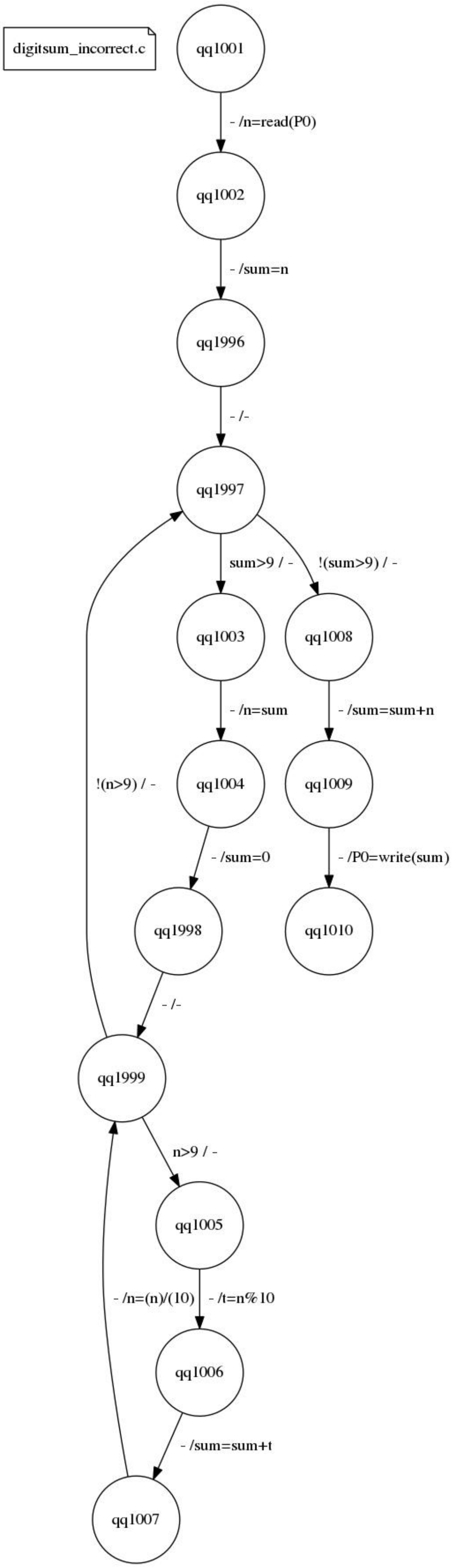


Figure B.5: Modified incorrect FSMD after introducing chain in loop2 in M_s for digitsum program.

It is to be noted that s_g is still at the state $qq1999$ in M_g and s_s is still at US. Now the next state from s_g is $qq1008$, which is on the outward edge of loop2, and which is along the loop1. The next state from s_s is $qq1008$ in M_s in figure B.5. As $qq1008$ is a non cut-point state in both the FSMDs, so chain copy mechanism comes into picture. It finds that there is only one transition $\langle -, sum = sum + n\rangle$ from $qq1008$ to the next cut-point $qq1997$ in M_g, which is also present after the next state $qq1008$ of s_s. This transition has to be copied to loop1 in M_s, hence it is removed from its present occurrence and moved inside loop1. This completes the correction mechanism as we get the resulting modified M_s same as the M_g. The figure B.6 shows the completely corrected M_s.

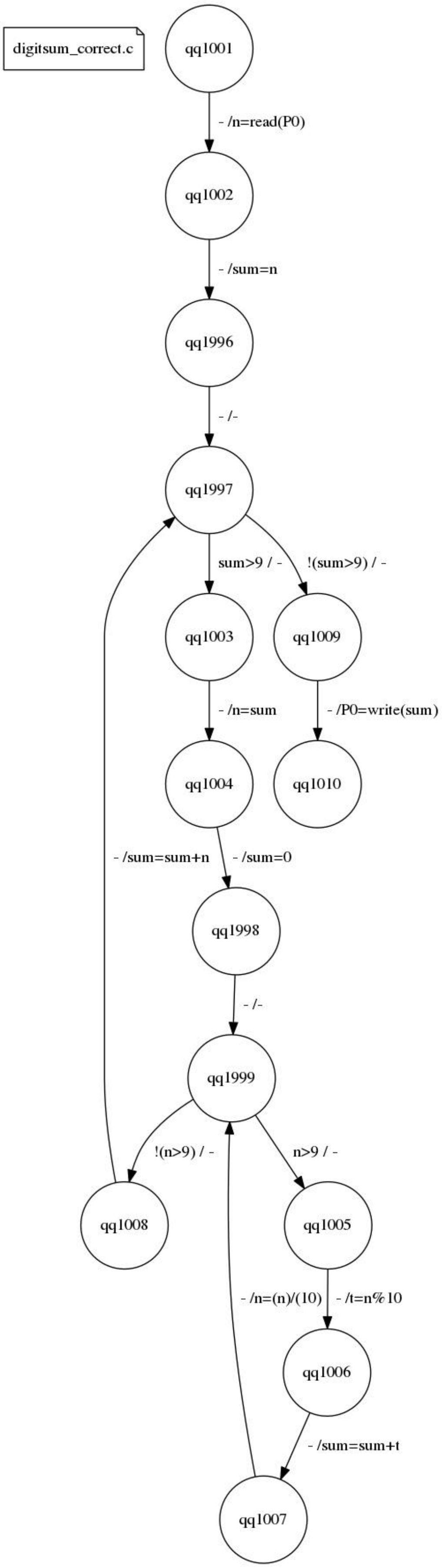


Figure B.6: Modified incorrect FSMD after completion of chain copy in loop1 in M_s for digitsum program.

Example B.2. The FSMDs and codes of golden and student's programs for calculating the points on a diamond are reproduced below (Figures B.7 and B.8). The program for this problem requires two loops. In student program both the loops are missing, making the student program inconsistent with teacher's program. We discuss below the simulation of the mechanism.

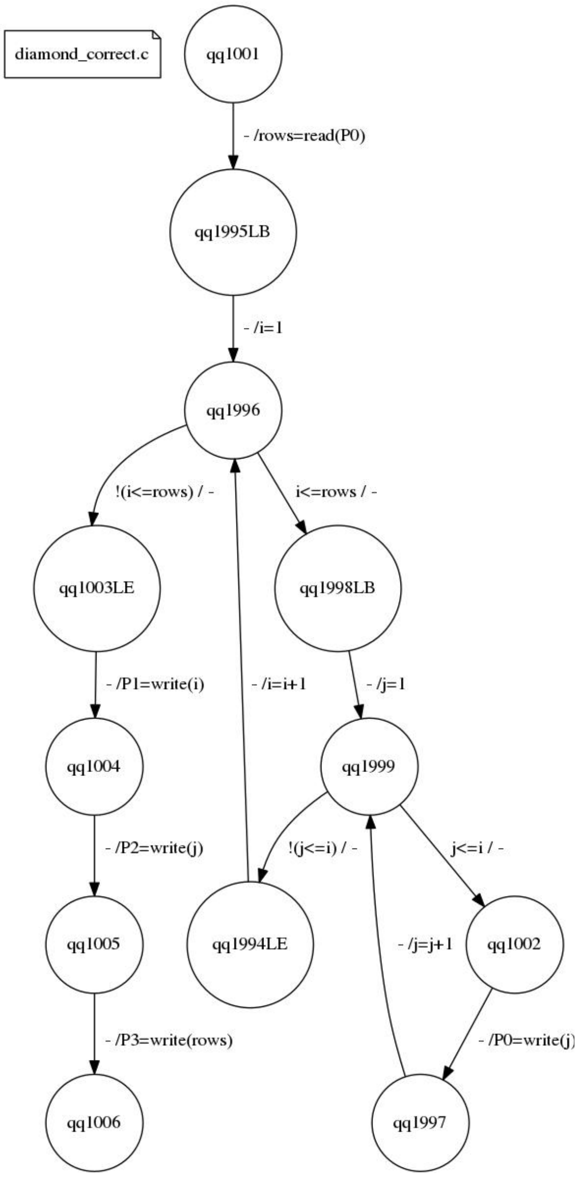


Figure B.7: M_g, diamond program

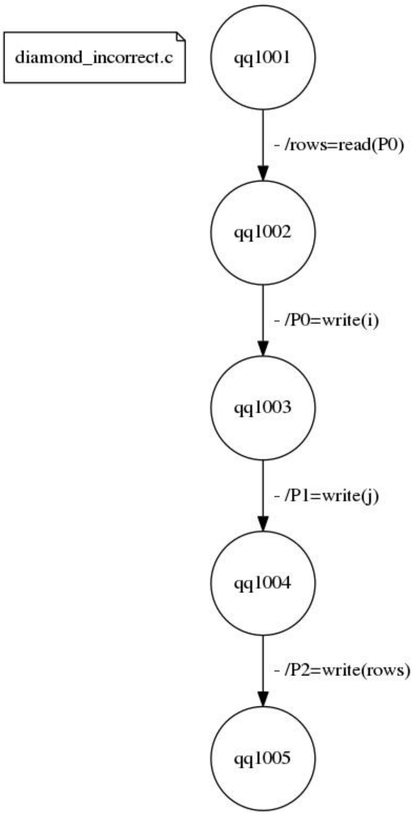


Figure B.8: M_s, diamond program

diamond_correct.c

```
#include <stdio.h>
int main() {
    int i, j, rows;
    printf("Enter number of rows: ");
    scanf("%d",&rows);
    for(i=1; i<=rows; ++i) {
        for(j=1; j<=i; ++j) {
            printf("%d", j);
        }
    }
    printf("%d", i);
    printf("%d", j);
    printf("%d", rows);

    return 0;
}
```

diamond_incorrect.c

```
#include <stdio.h>
int main() {
    int i, j, rows;
    printf("Enter number of rows: ");
    scanf("%d",&rows);
    printf("%d", i);
    printf("%d", j);
    printf("%d", rows);
    return 0;
}
```

We refer to the FSMDs in figures B.7 and B.8 in the following discussion. Execution of equivalence checker on these two FSMDs gives us $qq1001$ as the LCS. The CSLCS is the state $qq1001$. Containment checker takes the path p_s = $qq1001 \rightarrow qq1002 \rightarrow qq1003 \rightarrow qq1004 \rightarrow qq1005$, starting from $qq1001$ to next cut point i.e $qq1005$ in M_s. It tries to find whether this path is present in M_g or not. Containment checker outputs the path p_g = $qq1001 \rightarrow qq1995LB \rightarrow qq1996 \rightarrow qq1003LE \rightarrow qq1004 \rightarrow qq1005 \rightarrow qq1006$ of M_g, which is containing the path p_s in M_s. After the execution of step (3), we get the state $qq1002$ in M_s as US. The corresponding state in M_g, the state $qq1995LB$ is thus CSUS. We associate the variable state names s_g and s_s with CSUS and US respectively. DFS visit of M_g with s_g as root gives that it is neither the starting of a loop, nor a cut-point. So a trivial chain copy mechanism copies the transition $\langle -, i = 1 \rangle$ before US in the M_s. Now the s_g shifts down to the state $qq1996$ in M_g, which is the start of a loop state. In step (3) a self-loop, loop1, is introduced in M_s, corresponding to the loop at state s_g of M_g. This self-loop is introduced at a new state in M_s, just before the US (i.e., s_s). The condition of self-loop is kept the same as that of s_g. Now s_g moves down to the next state i.e., $qq1998LB$ in M_g. The s_s remains at the US. Figure B.9 shows the introduction of loop1.

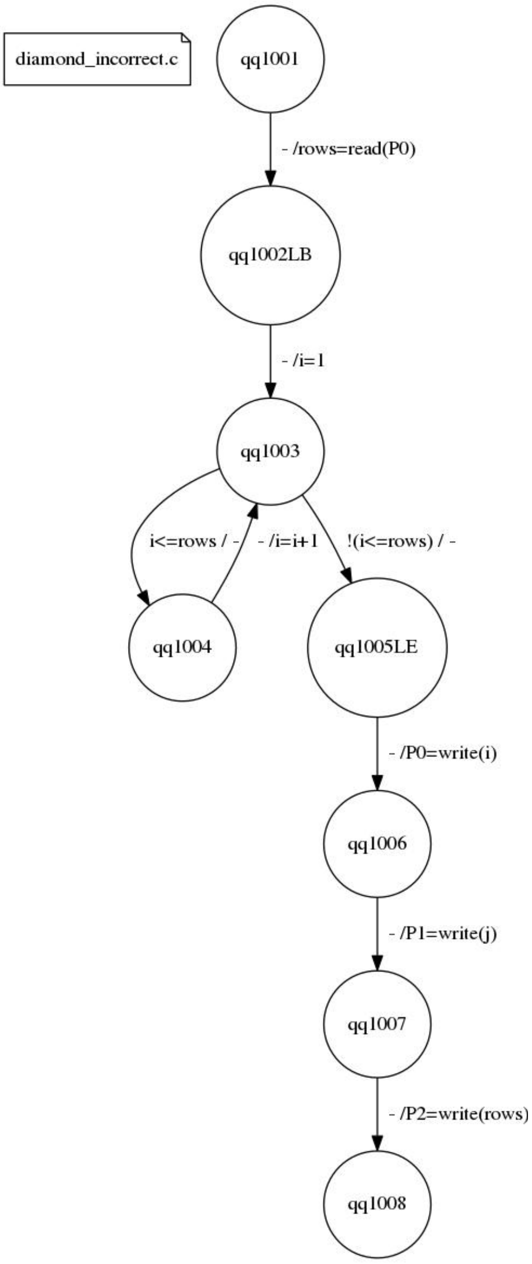


Figure B.9: Modified incorrect FSMD after introducing loop1 before the unmatched state in M_s for diamond program.

The current state where s_g is, is a state which is neither a cut-point nor a loop start state. Hence the next transition $\langle -, j = 1 \rangle$ is simply copied in the loop1 by the chain copying mechanism, as only this transition lies between the state s_g and the next cut-point on M_g. s_g then goes down to the next state $qq1999$, which is a loop start state. Hence a new self-loop, loop2 is introduced in the loop1. The condition of loop2 is same as the condition of s_g, i.e., $j <= i$. Insertion of self-loop loop2 is shown in figure B.10.

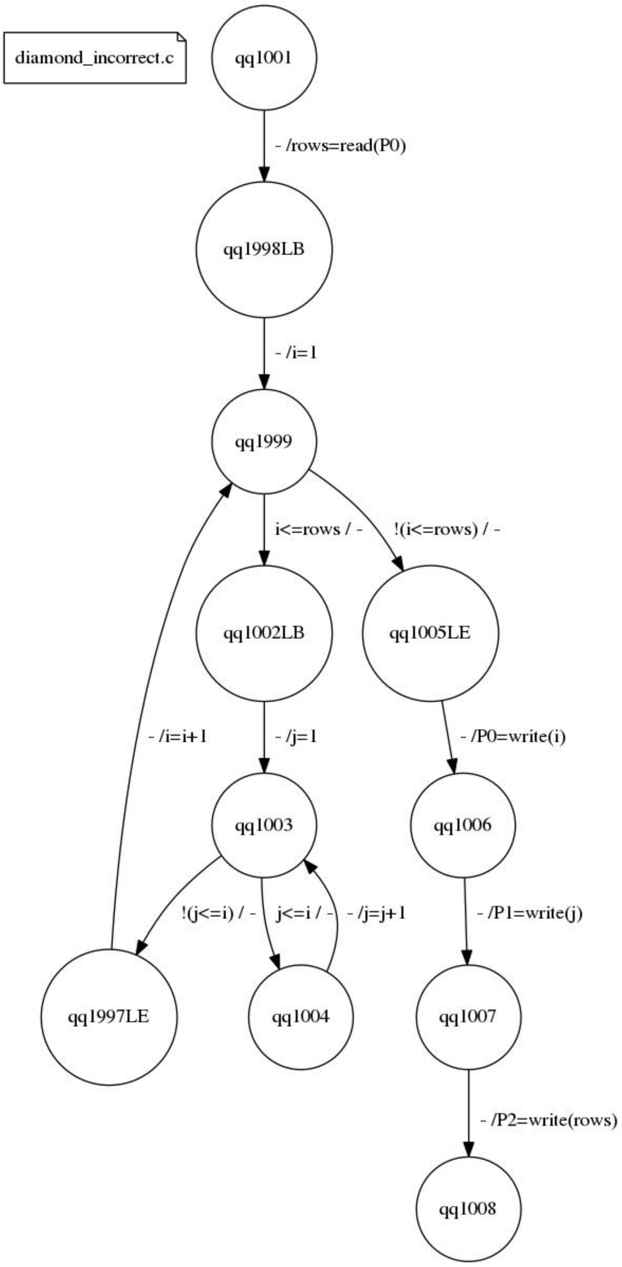


Figure B.10: Incorrect FSMD after inserting loop2 in M_s for diamond program.

s_g now moves to the next state $qq1002$. As $qq1002$ is a non cut-point and non loop start start state, so the path upto next cut-point $qq1999$ is put as ChainMg. This path not being the same as ChainMs, will be copied inside the loop2, thus completing the loop2 in M_s. This makes the modified M_s, the same as the FSMD M_g. The FSMD of modified student's program is shown in figure B.11.

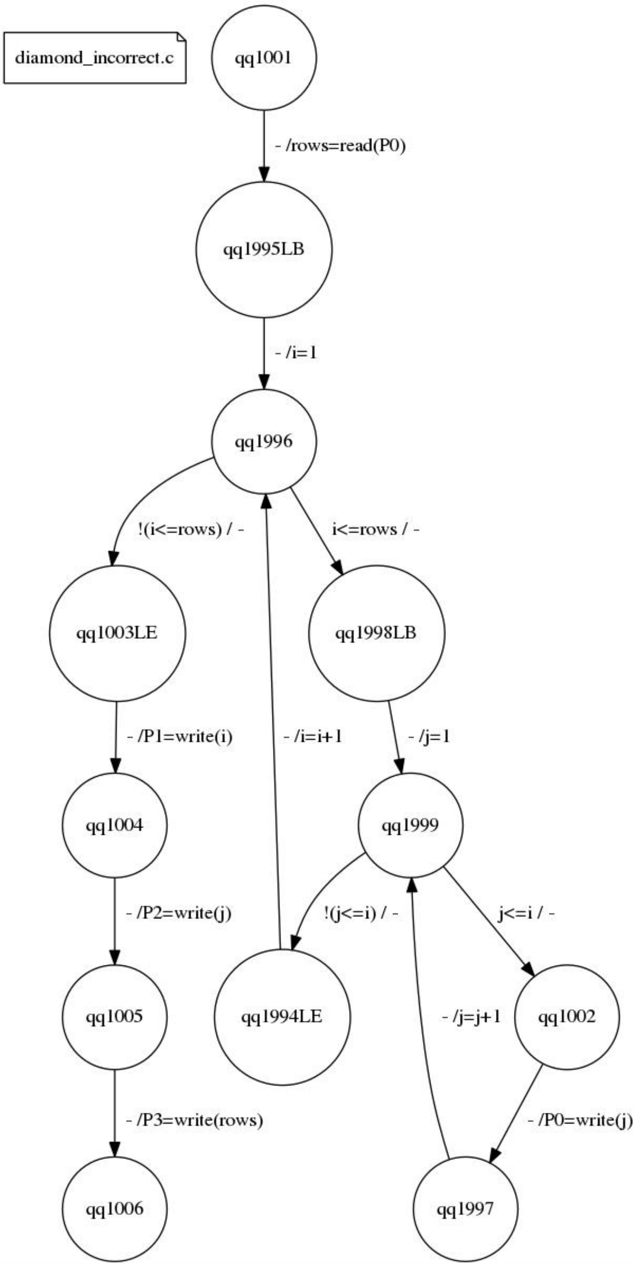


Figure B.11: Modified incorrect FSMD after introducing chain in loop2 in M_s for diamond program.

Example B.3. In this example we consider a program having no loops and no conditions. The golden program simple_correct.c and the student's program simple_incorrect are given below. Their respective FSMDs are shown in figures B.12 and B.13 respectively.

simple_correct.c

```
int main() {
     int i, j, k;
```

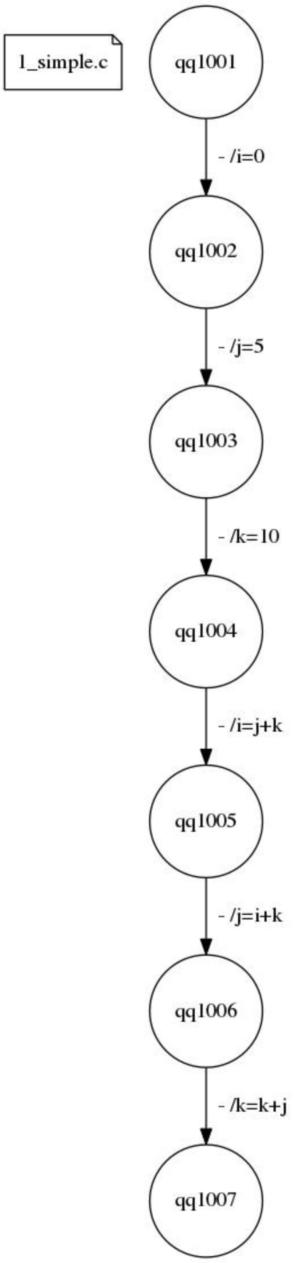


Figure B.12: M_g, a simple program

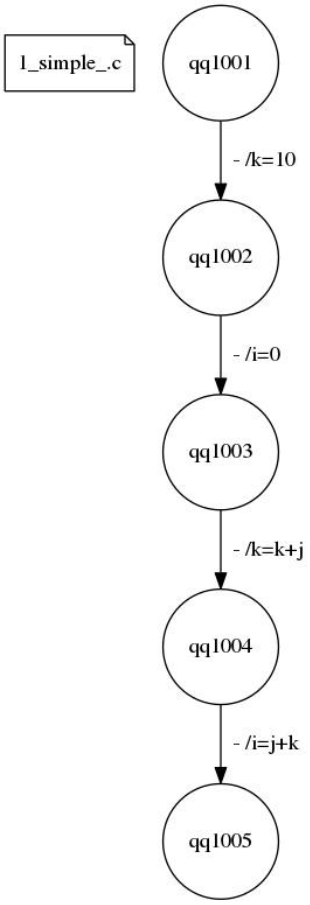


Figure B.13: M_s, a simple program

```
     i = 0;
     j = 5;
     k = 10;

     i = j + k;
     j = i + k;
     k = k + j;

     return 0;
}
```

simple_incorrect.c

```
#include <stdio.h>
int main() {
     int i, j, k;
     k = 10;
     i = 0;

     k = k + j;
     i = j + k;

     return 0;
}
```

Execution of equivalence checker on these two FSMDs gives us $qq1001$ as the LCS in M_s. The CSLCS is the state $qq1001$ in M_g. Containment checker takes the path $qq1001 \twoheadrightarrow qq1005$, starting from $qq1001$ to next cut point i.e $qq1005$ in M_s. It tries to find whether this path is present in M_g or not. Containment checker outputs the path $qq1001 \twoheadrightarrow qq1007$ of M_g, which is containing this path. After the execution of step (3), we get the state $qq1001$ in M_s as *unmatched state*, which is also s_s. The corresponding state, CSUS, in M_g is $qq1001$, thus s_g is also $qq1001$ in M_g. DFS visit of M_g with $qq1001$ (i.e., s_g) as root gives that $qq1001$ is neither the starting of loop, nor this is a cut-point. Hence, ChainMg is constructed from s_g to the next cut-point $qq1007$, in M_g. ChainMs is constructed from US to the next cut-point $qq1005$, in M_s. The chain copy mechanism now works on M_s as follows. Both the chains are compared and found unequal by Equate_chains function. ChainMg is, therefore, introduced in the M_s, transition after transition each time copying the transition if not already present in M_s and copying and replacing those transitions which are already present in M_s, but are in a different sequence. This way the entire ChainMg is copied to M_s, making M_s same as M_g.

Example B.4. The FSMDs and codes of golden and student's programs for calculating the sum of digits of the sum obtained until we get a single digit number are given below (Figures B.14 and B.15). The program for this problem requires two loops. In student program the inner while loop is missing, making the student program inconsistent with teacher's program. We discuss below the simulation of the mechanism.

digitsum_correct.c

```
#include<stdio.h>
int main() {
        int n, i, sum, t;
        printf("enter the number");
        scanf("%d", &n);
        sum = n;
        while (sum > 9) {
                n = sum;
                sum = 0;
                while (n > 9) {
                        t =n % 10;
                        sum = sum + t;
                        n = n / 10;
                }
                sum = sum + n;
        }
        printf("%d", sum);
        return 0;
}
```

digitsum_incorrect.c

```
#include<stdio.h>
int main() {
        int n, i, sum, t;
        printf("enter the number");
        scanf("%d", &n);
        sum = n;
        while (sum > 9) {
                n = sum;
                sum = 0;
                sum = sum + n;
        }
        printf("%d", sum);
        return 0;
}
```

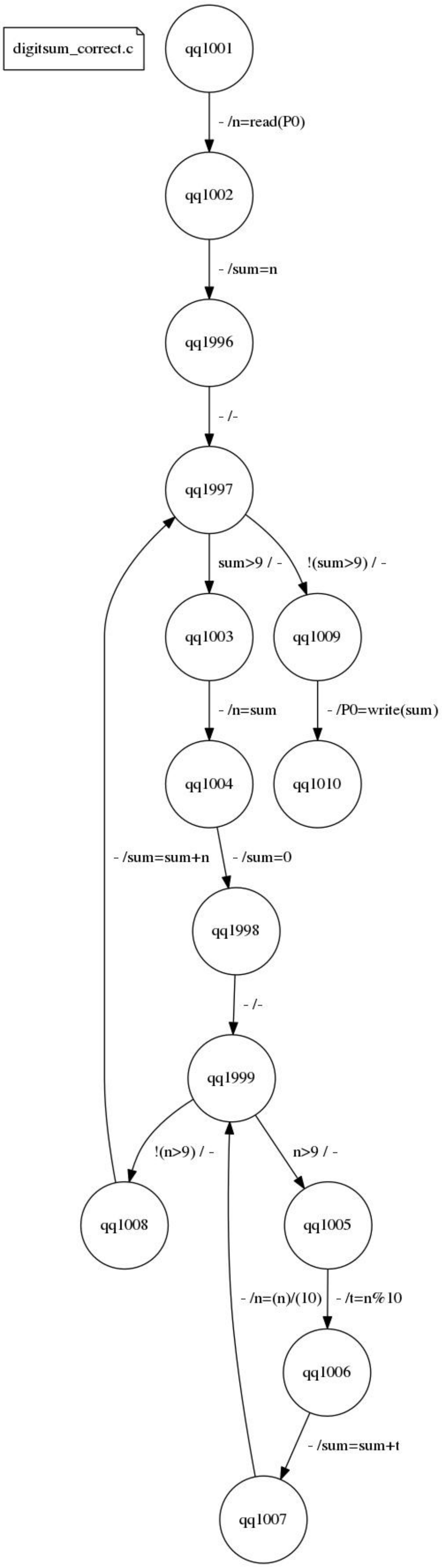


Figure B.14: M_g, digitsum program

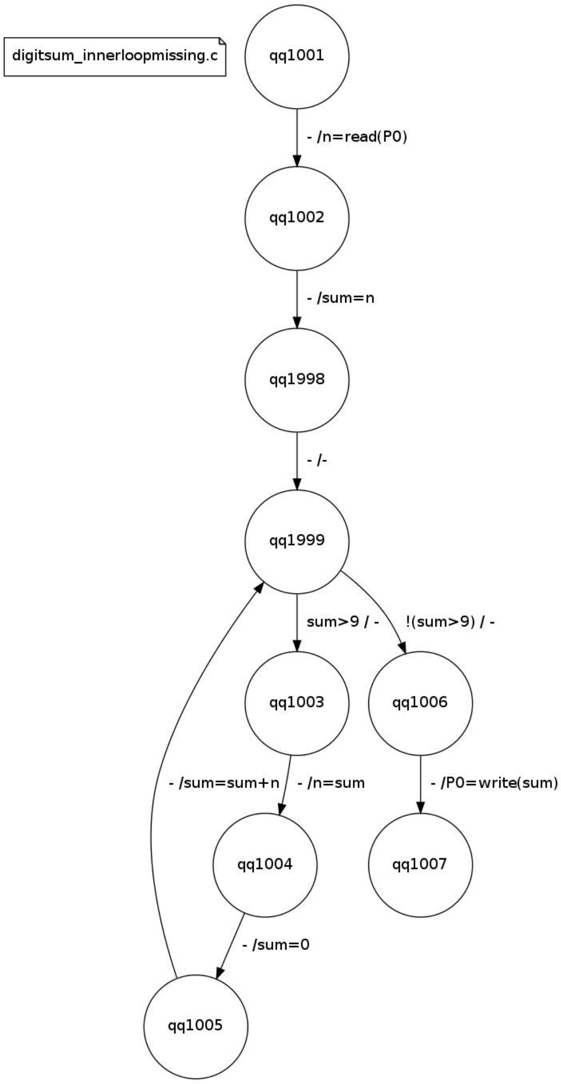


Figure B.15: M_s, digitsum program

Execution of equivalence checker on these two FSMDs gives us $qq1005$ as the LCS. The CSLCS is the state $qq1999$. Containment checker takes the path $p_s = qq1005 \rightarrow qq1999$, starting from $qq1005$ to next cut point i.e $qq1999$ in M_s. It tries to find whether this path is present in M_g or not. Containment checker outputs the path $p_g = qq1999 \rightarrow qq1008 \rightarrow qq1997$ of M_g, which is containing the path p_s in M_s. After the execution of step (3), we get the state $qq1005$ in M_s as US. The corresponding state in M_g, the state $qq1999$ is thus CSUS. We associate the variable state names s_g and s_s with CSUS and US respectively. DFS visit of M_g with s_g i.e.,$qq1999$ as root gives that $qq1999$ is the starting of a loop. In step (3) it introduces a self-loop, loop1, in M_s, corresponding to the loop at state $qq1999$ of M_g.

This self-loop is introduced at a new state $qq1006$ in M_s, in figure B.16, just before $qq1008$, the US (i.e., s_s). The condition of self-loop is kept the same as that of s_g. Now s_g moves down to the next state i.e., $qq1005$ in M_g. The s_s remains at the US. The modified code after this step becomes as follows. The modified FSMD of M_s is shown in figure B.16.

digitsum_innerloopmissingaddedselfloop.c

```
#include<stdio.h>
int main() {
        int n, i, sum, t;
        printf("enter the number");
        scanf("%d", &n);
        sum = n;
        while (sum > 9) {
                n = sum;
                sum = 0;
                while (n > 9) {
                }
                sum = sum + n;
        }
        printf("%d", sum);
        return 0;
}
```

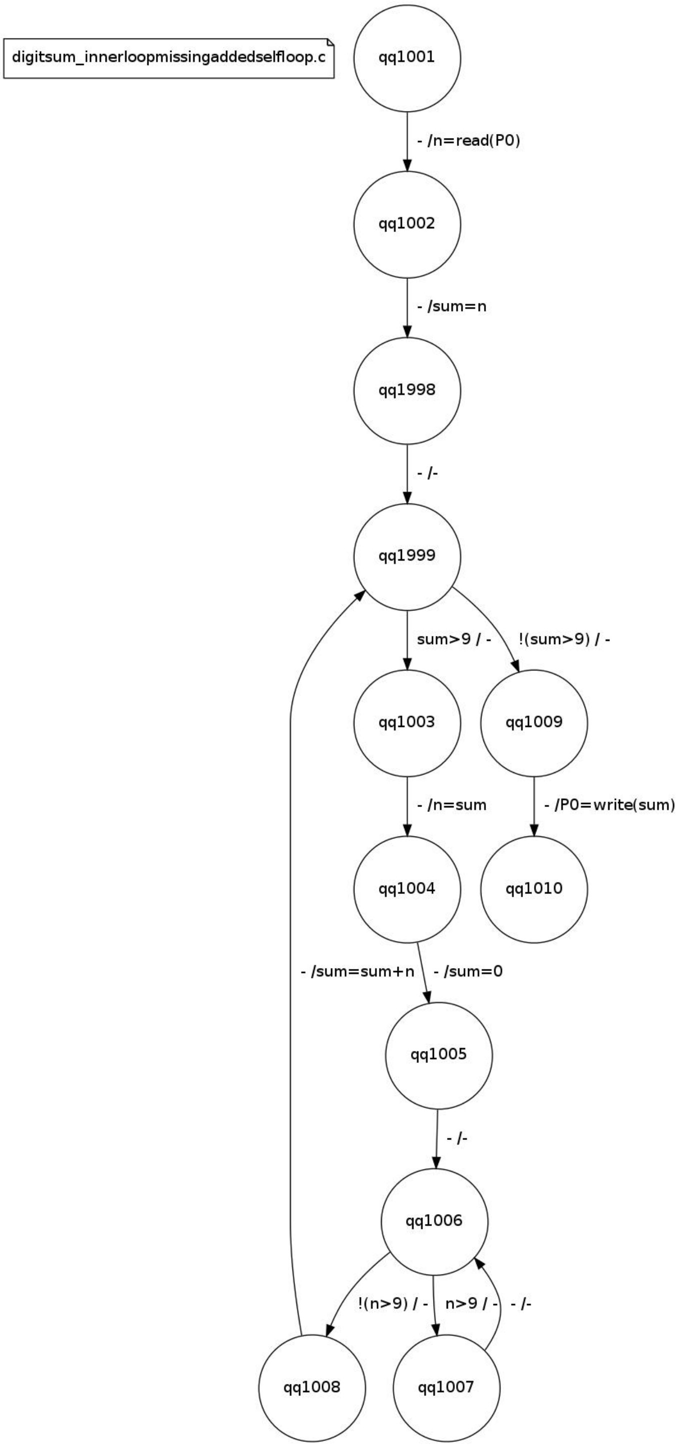


Figure B.16: Modified incorrect FSMD after introducing loop1 before the unmatched state in M_s for digitsum program.

Now s_g is at the state $qq1005$ and the s_s remains at the US, indicating that the FSMDs are equivalent upto s_g and s_s. As s_g is neither a loop nor a cut-point, so a chain, ChainMg, is made upto the next cut-point i.e. $qq1999$. A chain, ChainMs is made from s_s to the next cut-point, which is also s_s. The chain copy mechanism copies the chain in the loop1. The modified loop1 is shown in figure B.17, which makes M_s same as M_g. The modified code of the student's program becomes the following, which is same as the golden program. The FSMD of this program is shown in figure B.17.

digitsum_innerloopmissingfilledselfloop.c

```
#include<stdio.h>
int main() {
        int n, i, sum, t;
        printf("enter the number");
        scanf("%d", &n);
        sum = n;
        while (sum > 9) {
                n = sum;
                sum = 0;
                while (n > 9) {
                       t =n % 10;
                       sum = sum + t;
                       n = n / 10;
                }
                sum = sum + n;
        }
        printf("%d", sum);
        return 0;
}
```

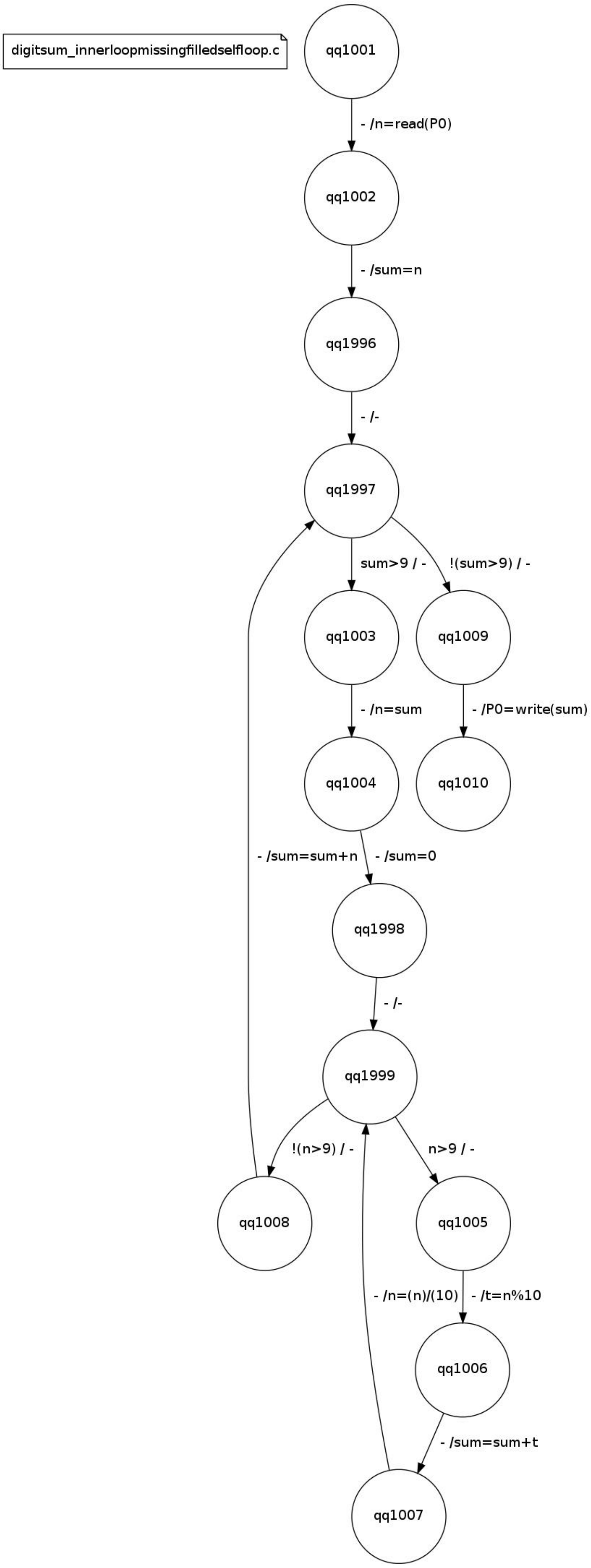


Figure B.17: Incorrect FSMD after modifying loop1 in M_s for digitsum program.

Appendix C

Outlines of a new procedure for correcting the student's programs

In this appendix, we provide the outlines of pseudocodes for the algorithms used to find out similarity of subgraphs in two given FSMDs and outline of an example to correct student's program by introducing missing code.

C.1 Introduction

The method presented here involves comparing the FSMDs of the golden and the student's program and correct the student's program by altering the student's program. Our attempt to correct should be such that it retains as much of student's code as possible. A program consists of loops, conditional constructs and straight line code. Our method identifies existence of loops, if on traversing FSMDs, the same state is encountered again. The conditional constructs are identified by a cut-point node, whose branches eventually meet at a common state, called join node. The notion of dominators has been used to identify join node. The conditional block are separated once we know the decision node and the join node. A backwards depth first traversal up to the decision node in a conditional construct leads to identify the edges in the conditional block. A conditional block can thus be segregated. In case of loops, back edges are identified using depth first search. The edges in the loop body can be identified by doing a reverse depth first traversal from the state at the tail of the back edge to the state at the head of the back edge. Loops are segregated using this method. While traversing a loop or a conditional construct, the respective handling functions are invoked, which are recursive in nature. Similarity between two paths in the two FSMDs is treated as the edit-distance between the statements occurring on the two paths. If same statements occur on the two paths, then the edit distance between the paths is treated to be zero.

In this appendix, an attempt has been made to develop a method to compare two FSMD's. In future we can improve our methods to compare even more generalized FSMD's with many loops and conditionals. We can also work on improving the space and time complexities of our methods.

C.2 Supporting procedures

The supporting procedures include the procedures for finding dominators, back-edges and finding join state of a conditional block etc., among others. An introduction to dominator node is as follows. A node *d* dominates a node *n* if every path from the entry node to *n* must go through *d*. By definition, every node dominates itself. A node *d* strictly dominates a node *n* if *d* dominates *n* and *d* does not equal *n*. The immediate dominator or *idom* of a node *n* is the unique node that strictly dominates *n* but does not strictly dominate any other node that strictly dominates *n*. Every node, except the entry node, has an immediate dominator. The dominance frontier of a node *d* is the set of all nodes *n* such that *d* dominates an immediate predecessor of *n*, but *d* does not strictly dominate *n*. It is the set of nodes where *d*'s dominance stops. A dominator tree is a tree where each node's children are those nodes it immediately dominates. Because the immediate dominator is unique, it is a tree.

```
Algorithm: findDominators
Input:startNode
Output:dom[]
Begin:
Set U;
dfs (U, startNode, NULL);
```

```
For each state v in U
    Set V;
    dfs (V, startnode, v); //exclude v
    Set K = U - V; //v dominates all elements of K
    For each k in K
        dom[k] = v ∪ dom[k];
return dom[]
```

Depth-first search (DFS) is an algorithm for traversing or searching tree or graph data structures. The algorithm starts at the root node (selecting some arbitrary node as the root node in the case of a graph) and explores as far as possible along each branch before backtracking.

Algorithm: `dfs`

Input:D, node, v // node: startnode / current node, v: node to be excluded

Output:

Begin:

```
If v ! = NULL and v==node //if v to be excluded and v is the node node
    return ;
Set D = D ∪ node;
For each successor x of node and x is not in D
    dfs(D, x, v);
```

Back-edge

Using the DFS algorithm a back-edge is found as an edge from a node to one of its ancestors in the dfs tree. For example in the FSMD shown in fig. C.1, only the following edge will be a back edge: `t10`.

Example: `findDominators`

Step 1: Initialize a set U.

Step 2: Apply dfs on Figure C.1 with start node q_0 and store all states in the set U.

$U = \{q_0, q_1, q_2, q_3, q_4, q_5, q_6, q_7, q_8, q_9, q_{10}\};$

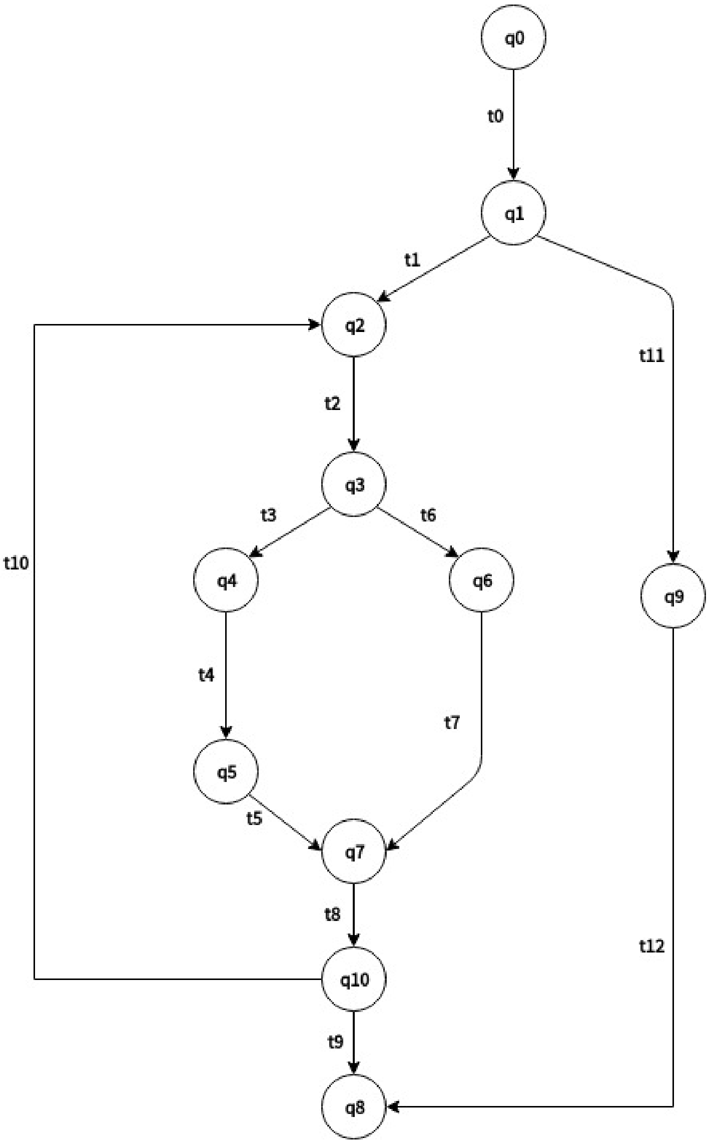


Figure C.1: FSMD of Golden Program, in the figure `t0..t11` are the set of statements, e.g. `t0={a=5;b=4;a=b+c;..}`

Step 3: For each state q_i , apply dfs from start node and store all nodes in V_i except the nodes through q_i .

Step 4: For each q_i calculate, $K_i = U - V_i$. Here, K_i contains all nodes that are dominated by q_i .

Step 5: Store the dominators of each node q_i in $dom[\ q_i\ ]$.

Join state

Algorithm: `findJoinState`

Input: S_g

Output: S_{temp}

Begin:

```
Let $t_g$ be an outgoing transition from $S_g$ and $t_g$ is not a backedge
If incoming transitions to $t_g^f$ >1 and $S_g$ in dominators($t_g^f$)
    return $t_g^f$;
Hashtable visited;
insert $t_g^f$ to visited
$S_{temp}$ = ($t_g^f$)
Let $t_g$ be an outgoing transition from $S_{temp}$ and $t_g$ is not a backedge;
    $S_{temp}$ = ($t_g^f$)
Let $t_g$ be an outgoing transition from $S_{temp}$ and $t_g$ is not a backedge;
For each x in visited and x not in dominators($S_{temp}$)
    insert $S_{temp}$ to visited
    $S_{temp}$ = ($t_g^f$)
    Let $t_g$ be an outgoing transition from $S_{temp}$ and $t_g$ is not a backedge;
return $S_{temp}$
```

In FSMD, a path between two nodes is a possible ordering of execution of the statements from start node to end node.

Algorithm: `findAllPaths`

Input:startState, endState, Path[],AllPaths[][]

Output:

Begin:

Table C.1: Dominator table

q_i	V_i	K_i=$U - V_i$	$dom[q_i]$
q0	ξ	q0,q1,q2,q3,q4,q5,q6,q7,q8,q9,q10	q0
q1	q0	q1,q2,q3,q4,q5,q6,q7,q8,q9,q10	q0,a1
q2	q0,q1,q8,q9	q0,q2,q3,q4,q5,q6,q7,q10	q0,q1,q2
q3	q0,q1,q2,q8,q9	q3,q4,q5,q6,q7,q10	q0,q1,q2,q3
q4	q0,q1,q2,q3,q6,q7,q8,q9,q10	q4,q5	q0,q1,q2,q3,q4
q5	q0,q1,q2,q3,q4,q6,q7,q8,q9,q10	q5	q0,q1,q2,q3,q4,q5
q6	q0,q1,q2,q3,q4,q5,q7,q8,q9,q10	q6	q0,q1,q2,q3,q6
q7	q0,q1,q2,q3,q4,q5,q6,q8,q9	q7,q10	q0,q1,q2,q3,q7
q8	q0,q1,q2,q3,q4,q5,q6,q7,q9,q10	q8	q0,q1,q8
q9	q0,q1,q2,q3,q4,q5,q6,q7,q8,q10	q9	q0,q1,q9
q10	q0,q1,q2,q3,q4,q5,q6,q7,q8,q9	q10	q0,q1,q2,q3,q7,q10

```
If startState is equal to endState
    add Path into allPaths
    return ;
For each outgoing transition ti from startState
    If $t_i$ not in Path
        add $t_i$ to Path.
        call recursively findAllPaths with input $t_i^f$, endState, Path, AllPaths
        delete $t_i$ from the Path
```

Edit-distance is the cost to convert a string to another string with minimal changes. As string is the ordering of the characters, a path is the ordering of edges and in FSMD each edge has some statement, so paths are the ordering of statements. Considering each statement as a character, we can find the edit distance between two paths.

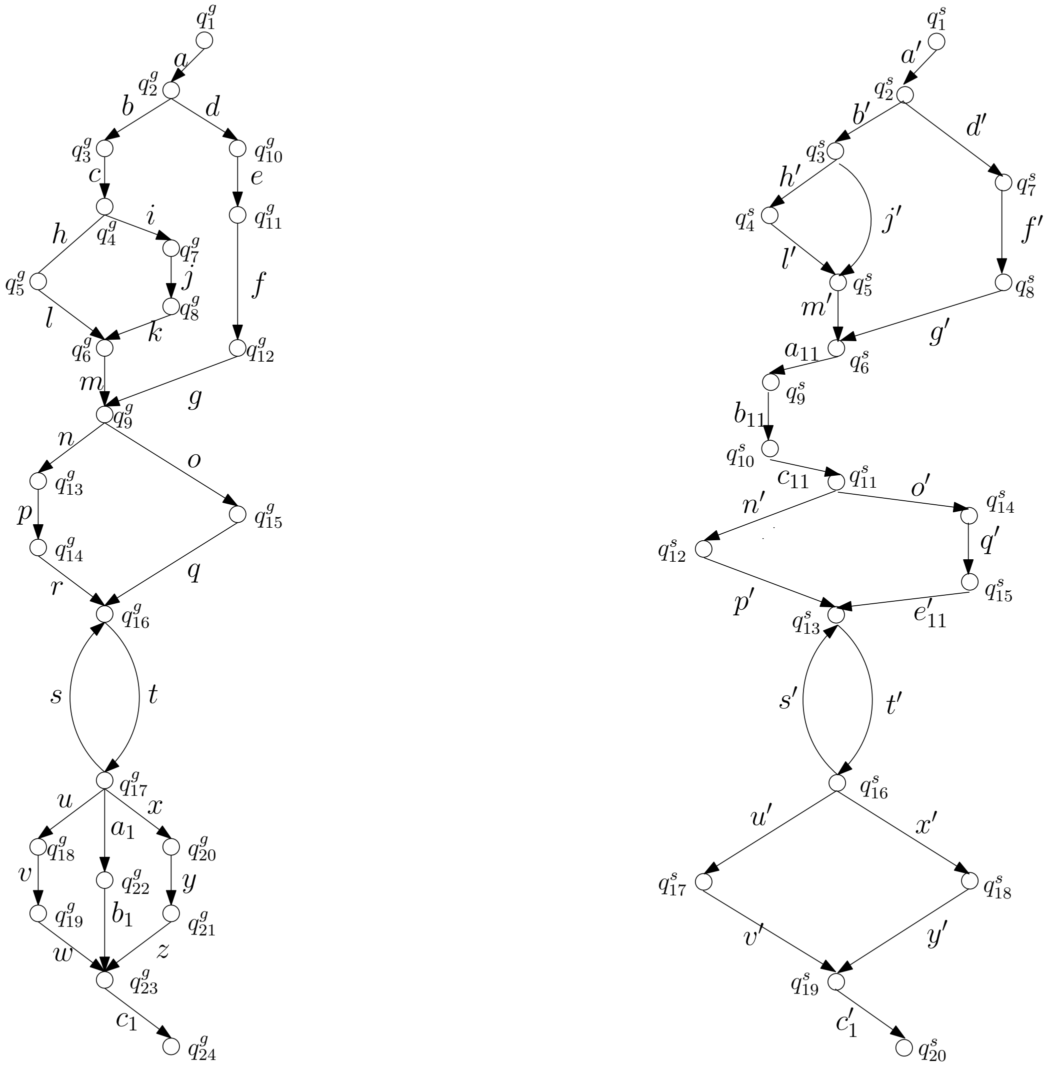
Algorithm: **`minEditDistanceDP`**

Input: str_1[],str_2[],m,n

Output: cost

Begin:

```
dp[m+1][n+1];//Create a table to store results of sub-problems
 For i in range(0,m-1) //each character in $str_1$
     For j in range(0,n-1)//each character in $str_2$
         If i=0
             dp[i]][j]=j
         Else If j=0
             dp[i][j] = i;
         Else If $str_1$[i-1] == $str_2$[j-1]
             dp[i][j] = dp[i-1][j-1];
         Else
             dp[i][j] = min (dp[i][j-1] + InsertCost, dp[i-1][j] + RemoveCost,

                   dp[i-1][j-1] + ReplaceCost);
cost=dp[m][n];
return cost;
```



FSMD of golden program

FSMD of student's program

Figure C.2: Figure with FSMDs for various examples.

segregate_loop

The pseudocode for this program is given as follows.

Listing C.1: segregate_loop program

```
Marked_loop_Transitions[] segregate_loop(sg, backedge tbg){
       mark tbg and store it in an array Marked_loop_Transitions[];
       if(sg == tbg_start){//tbg_start is tail node of the back edge.
              return;
```

```
        }
        else{
                For each incoming transition ti to tbg_start{
                        if(ti is not marked){
                                segregate_loop(sg, ti);
                        }
                }//end_for
        }//end_else
        return Marked_loop_Transitions[];
}//end_fn
```

In the following description of steps of the above pseudocode, M stands for the array $Marked\_loop\_Transitions[\,]$. The fig C.2 has been referred in the underlying description.

FSMD of golden program

- $s_g = q^g_{16}$; backedge $tbg = s$;
- Mark tbg and store it in array $M[\,]$; $M = \{s\}$;
- $tbg\_start = q^g_{17}$;
- For each incoming transition t_i to tbg_start, if it is not marked, call $segregate\_loop(q^g_{16}, t_i)$
- It will terminate when s_g will be equal to tbg_start.
- Finally M will contain edges s and t

  $M = \{s, t\}$
- Return M.

FSMD of student's program

- $s_g = q^s_{13}$; backedge $tbg = s'$;
- Mark s' and store it in array M. $M = \{s'\}$.

- $tbg\_start = q^s_{16}$.
- For each incoming transition t_i to tbg_start, if it is not marked, call $segregate\_loop(q^s_{13}, t_i)$.
- It will terminate when s_g will be equal to tbg_start.
- $M = \{s', t'\}$.
- Return M.

segregate_conditional

The pseudocode for this program is given as follows.

Listing C.2: segregate_conditional program

```
Marked_conditional_Transitions[] segregate_conditional(sg, sj){
        //sg=entry state for conditional
        //sj=current state(initially join state)
        if(sg == sj){
                return;
        }
        else{
                For each incoming transition ti to sj{
                        if(ti is not marked){
                                mark ti and store it in an array
                                Marked_conditional_Transitions[];
                                segregate_conditional(sg, ti_start);
                        }
                }//end_for
        }//end_else
        return Marked_conditional_Transitions[];
}//end_fn
```

In the following description of steps of the above pseudocode, M stands for the array $Marked\_conditional\_Transitions[\,]$. The fig C.2 has been referred in the underlying description.

FSMD of golden program

- $sg = q_2^g$; $sj = q_9^g$ (join point).
- For each incoming transition ti to sj, if ti is not marked, mark it and store it in M. $M = \{m\}$.
- Call $segregate\_conditional(q_2^g, q_9^g)$. $M = \{m, l\}$.
- Similarly, $segregate\_conditional()$ will be called recursively.
- Finally, $M1 = \{m, l, h, c, b, i, j, k, g, f, e, d\}$.
- $M2 = \{l, k, h, j\}$.
- $M3 = \{r, q, p, n, o\}$.
- $M4 = \{w, v, u, z, y, x, b_1, a_1\}$.

FSMD of student's program

- $M1 = \{m', l', h', b', j', g', f', d'\}$.
- $M2 = \{l', h', j'\}$.
- $M3 = \{p', n', e'_{11}, q', o'\}$.
- $M4 = \{v', u', y', x'\}$.

segregate_graph

The pseudocode for this program is given as follows.

Listing C.3: segregate_graph program

```
segregate_graph (State sg, Set backedges, visited) {
       visited.insert(sg);
       int loop_found = 0;
       For each incoming transition ti on sg {
              if(ti in backedges) {
                segregate_loop(sg, ti);
                loop_found = 1;
            }
        }
        if (loop_found == 0) {
            if (sg is a cut-point) {
                State sj = find_join_state (sg);
                segregate_conditional (sg, sj);
            }
        }
        For each outgoing transition t_temp from sg {
            if(t_temp_f not in visited)
                segergate_graph(t_temp_f, backedges, visited);
        }
}
```

- In the following description of steps of the above pseudocode, M stands for the array $Marked\_conditional\_Transitions$[] and L stands for the array $Marked$ $\_loop\_Transitions$[].

- The function segregate_graph() will call the functions segregate_loop() and segregate_conditional(), depending upon the node type.

- If the current node is a conditional, then it will call segregate_conditional() and give FSMDs of all the conditionals and if node has backedges, then it calls segregate_loop() and get FSMDs of all the loops within the given FSMD.

FSMD of golden program

We get the loops and conditionals as follows.

- **Loops**:

  $L1 = \{s, t\}$.

- **Conditionals**:

- $M1 = \{l, k, h, j, i\}$
- $M2 = \{m, l, h, c, b, i, j, k, g, f, e, d\}$.
- $M3 = \{r, q, p, n, o\}$.
- $M4 = \{w, v, u, z, y, x, b_1, a_1\}$

FSMD of student's program

We get the following loops and conditionals by applying *segergate_graph* on the FSMD of student's program.

- **Loops**:

  $L1 = \{s', t'\}$.

- **Conditionals**:

  - $M1 = \{m', l', h', j', g', f', d', b'\}$.
  - $M2 = \{l', h', j'\}$.
  - $M3 = \{p', h', e'_{11}, q', o'\}$.
  - $M4 = \{v', u', y', x'\}$.

FSMD of golden program

U = $\{q^g_1, q^g_2, q^g_3, q^g_4, q^g_5, q^g_6, q^g_9, q^g_{13}, q^g_{14}, q^g_{16}, q^g_{17}, q^g_{18}, q^g_{19}, q^g_{23}, q^g_{24}, q^g_{22}, q^g_{20}, q^g_{21}, q^g_{15}, q^g_7, q^g_8, q^g_{10}, q^g_{11}, q^g_{12}\}$.

- After performing dfs on the golden FSMD, set U will contain all the possible states.
- For each state *s_temp* in U and for every transition *t_temp*, check if *t_temp_f* is present in the dominator set of *t_temp_s*.
- In the given example, we can observe that the state q^g_{16} is present in the dominator set of q^g_{17}.

  $dom[q^g_{17}] = \{q^g_1, q^g_2, q^g_9, q^g_{16}\}$

  $q^g_{16} \in dom[q^g_{17}]$

- So, the edge s is a back-edge. Insert s in the set of back-edges and return.

FSMD of student's program

$U = \{q^s_1, q^s_2, q^s_3, q^s_4, q^s_5, q^s_6, q^s_9, q^s_{10}, q^s_{11}, q^s_{12}, q^s_{13}, q^s_{16}, q^s_{17}, q^s_{19}, q^s_{20}, q^s_{18}, q^s_{14}, q^s_{15}, q^g_7, q^g_8\}$.

- Proceeding in the same way as above, we can observe that $q_{13} \in \mathrm{dom}[q^s_{16}]$.

  $dom[q^s_{16}] = \{q^s_1, q^s_2, q^s_6, q^s_9, q^s_{10}, q^s_{11}, q^s_{13}\}$

  $q_{13} \in dom[q^s_{16}]$

- So, the edge s' is a backedge.

C.3 Outline of an overall correction scheme

To illustrate the overall correction scheme, we consider the FSMDs shown in figure C.3. The scheme presented below is a recursive scheme and works as follows.

1. Find the back-edges in the two FSMDs and for each back-edge from golden program, find its least cost counterpart in the FSMD of student's program.(this step is not shown in figure C.3.

2. Segregate loops attached to the pairs of back-edges found above, e.g., L_1, L_2 in M_G and L'_1, L'_2 in M_S.

3. Make a copy of the loops L_1, $L2$, L'_1 and $L'2$.

4. Work on the copies to get the dissimilarity cost between the pairs (L_1, L'_1), (L'_1, L'_2), (L_2, L'_1) and (L_2, L'_2).

5. Finally select the pairs for L_1 and L_2, which have the least cost, e.g., let us assume (L_1, L'_2) and (L_2, L'_1) have maximum similarity (least cost of correction).

6. Similarly segregate the conditional blocks C_1, C_2, C'_1 and C'_2. Make their copies. Using the copies, select the pairs for C_1 and C_2, which have maximum similarity, e.g., let us assume (C_1, C'_2) and (C_2, C'_1) have the maximum similarity.

7. Recursively correct the matched loops and the conditional blocks and obtain the block correction cost C_B. To correct loops L and L', remove their back-edges making their tail nodes corresponding states and then apply this algorithm recursively on rest of the loop bodies of L and L'.

8. All the loop entry/exit states of the least cost pairs obtained as above, become corresponding states.

9. All the branch entry/exit states of the least cost pairs obtained as above, become corresponding states. In the figure C.3, states S_i and S'_i become the corresponding states.

10. Each matched and corrected conditional block is replaced with a trivial edge without any operation, between the block entry and exit states.

11. Each matched and corrected loop is removed leaving behind only the loop entry/exit node.

12. On the reduced FSMD obtained after the previous two operations, apply the algorithm for finding all paths between successive corresponding points and match up those paths according to their similarities and apply corrections according to the minimum edit-distance computations. Let this cost of correction be C_W.

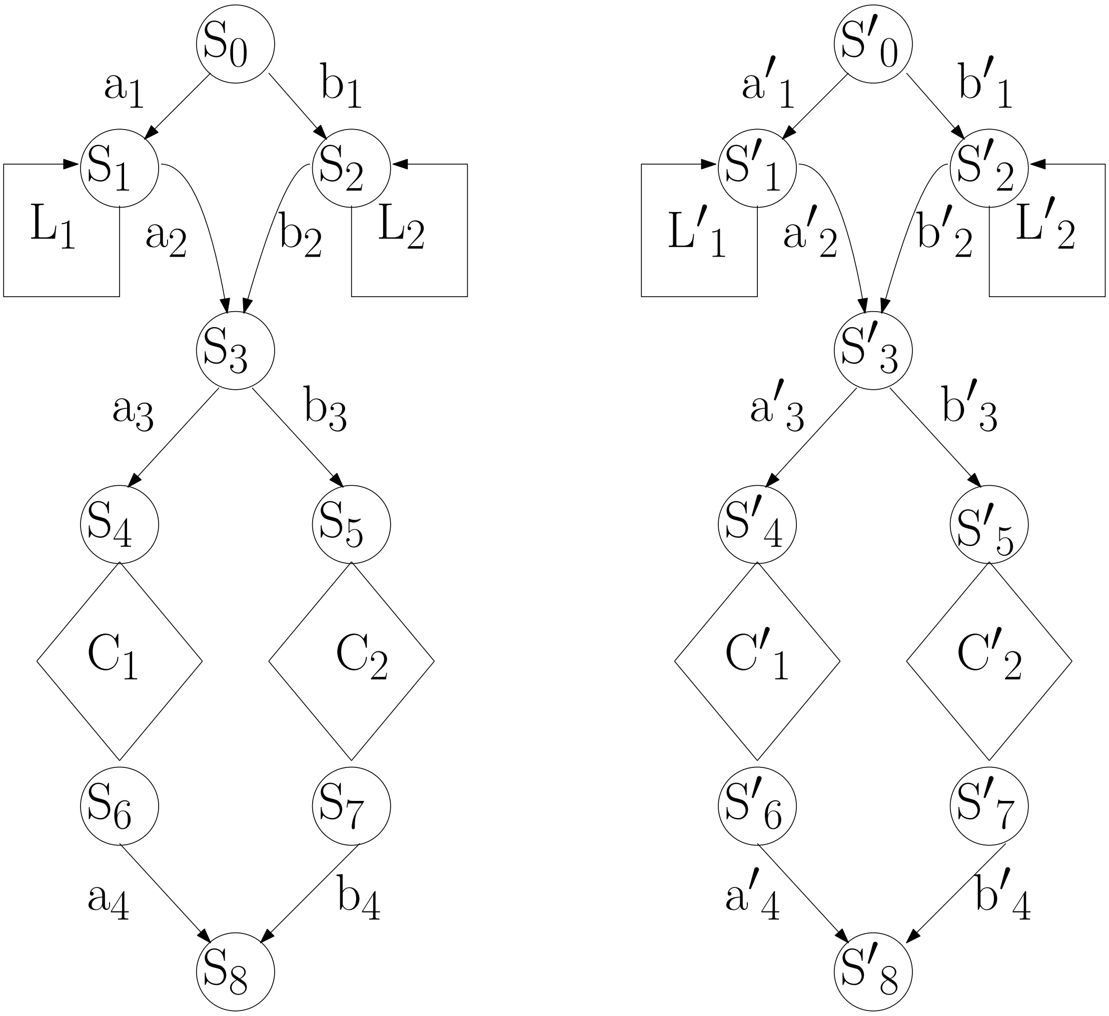
13. Total cost for correction is $(C_B + C_W)$. This cost is returned.

The algorithm for measuring similarity is given below.

```
Algorithm: similarityMeasure
Input: N_s (start point in student's FSMD)
       NC_s (Next cut-point in student's FSMD)
       N_g (start point in golden FSMD)
       NC_g (Next cut-point in golden FSMD)
Output: similarityCost
Begin:
1: for each path P_s between NC_s and N_s do
2:     for each path P_g between NC_g and N_g do
```

```
similarityCost=minEditDistanceDP( $P_g$ ,$P_s$ );
```

The above outline of the correction procedure does not cover some corner cases such as a missing block or missing loop, those have to be incorporated to make the algorithm complete.



(a) M_G
FSMD of golden program

(b) M_S
FSMD of student's program

Figure C.3: Figure with FSMDs for finding similarity and overll correction.

Bibliography

[1] CVC4 - the smt solver. http://cvc4.cs.nyu.edu/web/.

[2] GCC, the GNU Compiler Collection. https://gcc.gnu.org/.

[3] MOOC List. http://www.mooc-list.com/.

[4] Virtual Labs. http://www.vlab.co.in/.

[5] A. Adam and J.-P. H. Laurent. Laura, a system to debug student programs. *Artificial Intelligence*, 15(1-2):75–122, 1980.

[6] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers principles, techniques, and tools*. Addison-Wesley, Reading, MA, 1986.

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

[8] C. Alippi. *Randomized Algorithms*, pages 53–93. Springer International Publishing, Cham, 2014.

[9] M. Amelung, K. Krieger, and D. Rosner. E-assessment as a service. *IEEE Transactions on Learning Technologies*, 4:162–174, 2011.

[10] K. Banerjee. *Translation Validation of Optimizing Transformations of Programs using Equivalence Checking*. PhD thesis, IIT Kharagpur, India, 2015.

[11] K. Banerjee, C. Karfa, D. Sarkar, and C. Mandal. A value propagation based equivalence checking method for verification of code motion techniques. In *ISED*, pages 67–71, 2012.

[12] K. Banerjee, C. Karfa, D. Sarkar, and C. Mandal. Verification of code motion techniques using value propagation. *IEEE Trans. on CAD of ICS*, 33(8):1180–1193, 2014.

[13] K. Banerjee, C. Mandal, and D. Sarkar. Extending the scope of translation validation by augmenting path based equivalence checkers with smt solvers. In *VLSI Design and Test, 18th International Symposium on*, pages 1–6, July 2014.

[14] K. Banerjee, D. Sarkar, and C. Mandal. Extending the FSMD framework for validating code motions of array-handling programs. *IEEE Trans. on CAD of ICS*, 33(12):2015–2019, 2014.

[15] S. Benford, E. K. Burke, E. Foxley, and C. A. Higgins. The ceilidh system for the automatic grading of students on programming courses. In *Proceedings of the 33rd annual on Southeast regional conference*, pages 176–182. ACM, 1995.

[16] V. Bentkus. On HoeffdingâĂŹs inequalities. *Ann. Probab.*, 32(2):1650–1673, 04 2004.

[17] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *Proceedings of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, TACAS '99, pages 193–207, Berlin, Heidelberg, 1999. Springer-Verlag.

[18] M. Blumenstein, S. Green, S. Fogelman, A. Nguyen, and V. Muthukkumarasamy. Performance analysis of game: A generic automated marking environment. *Computers & Education*, 50(4):1203 – 1216, 2008.

[19] M. Blumenstein, S. Green, A. Nguyen, and V. Muthukkumarasamy. Game: a generic automated marking environment for programming assessment. In *International Conference on Information Technology: Coding and Computing, 2004. Proceedings. ITCC 2004.*, volume 1, pages 212–216 Vol.1, April 2004.

[20] A. R. Bradley and Z. Manna. *The Calculus of Computation: Decision Procedures with Applications to Verification.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2007.

[21] B. Cheang, A. Kurnia, A. Lim, and W.-C. Oon. On automated grading of programming assignments in an academic institution. *Computers & Education*, 41(2):121 – 131, 2003.

[22] P. M. Chen. An automated feedback system for computer organization projects. *IEEE Transactions on Education*, 47(2):232–240, May 2004.

[23] H. Chernoff. *Conservative bounds on extreme P-values for testing the equality of two probabilities based on very large sample sizes*, volume Volume 45 of *Lecture Notes–Monograph Series*, pages 250–254. Institute of Mathematical Statistics, Beachwood, Ohio, USA, 2004.

[24] E. Clarke, D. Kroening, and F. Lerda. A tool for checking `ANSI-C` programs. In K. Jensen and A. Podelski, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 168–176, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[25] L. Cordeiro, B. Fischer, and J. Marques-Silva. Smt-based bounded model checking for embedded ansi-c software. In *2009 IEEE/ACM International Conference on Automated Software Engineering*, pages 137–148, Nov 2009.

[26] C. Daly and J. M. Horgan. An automated learning system for java programming. *IEEE Transactions on Education*, 47(1):10–17, Feb 2004.

[27] C. Douce, D. Livingstone, and J. Orwell. Automatic test-based assessment of programming: A review. *J. Educ. Resour. Comput.*, 5(3), Sept. 2005.

[28] P. J. Downey, R. Sethi, and R. E. Tarjan. Variations on the common subexpression problem. *J. ACM*, 27(4):758–771, Oct. 1980.

[29] C. C. Ellsworth, J. B. Fenwick, Jr., and B. L. Kurtz. The quiver system. In *Proceedings of the 35th SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '04, pages 205–209, New York, NY, USA, 2004. ACM.

[30] E. Enström, G. Kreitz, F. Niemelä, P. Söderman, and V. Kann. Five years with kattis - using an automated assessment system in teaching. In *2011 Frontiers in Education Conference (FIE)*, pages T3J–1–T3J–6, Oct 2011.

[31] J. Ferrante and C. W. Rackoff. *The computational complexity of logical theories*, volume 718. Springer, 2006.

[32] R. W. Floyd. Assigning meanings to programs. *Proceedings of Symposium on Applied Mathematics*, 19:19–32, 1967.

[33] D. D. Gajski, N. D. Dutt, A. C. Wu, and S. Y. Lin. *High-Level Synthesis: Introduction to Chip and System Design*. Kluwer Academic, 1992.

[34] M. A. Ghodrat, T. Givargis, and A. Nicolau. Expression equivalence checking using interval analysis. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 14(8):830–842, Aug 2006.

[35] D. Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Comput. Surv.*, 23(1):5–48, Mar. 1991.

[36] R. Hammack. *Book of Proof*. Virginia Commonwealth University, Math Department, 2nd edition, 2013.

[37] C. Higgins, T. Hegazy, P. Symeonidis, and A. Tsintsifas. The coursemarker cba system: Improvements over ceilidh. *Education and Information Technologies*, 8(3):287–304, 2003.

[38] P. Ihantola, T. Ahoniemi, V. Karavirta, and O. Seppälä. Review of recent systems for automatic assessment of programming assignments. In *Koli Calling*, pages 86–93, 2010.

[39] D. Jackson and M. Usher. Grading student programs using assyst. In *Proceedings of the Twenty-eighth SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '97, pages 335–339, New York, NY, USA, 1997. ACM.

[40] M. Joy, N. Griffiths, and R. Boyatt. The boss online submission and assessment system. *ACM Journal of Educational Resources in Computing*, 5(3), 2005.

[41] L. C. Kaczmarczyk, E. R. Petrick, J. P. East, and G. L. Herman. Identifying student misconceptions of programming. In *Proceedings of the 41st ACM Technical Symposium on Computer Science Education*, SIGCSE '10, pages 107–111, New York, NY, USA, 2010. ACM.

[42] W. Kahan. Pracniques: further remarks on reducing truncation errors. *Commun. ACM*, 8(1):40, 1965.

[43] C. Karfa. Hand-in-hand verification and synthesis of digital circuits. Master's thesis, IIT Kharagpur, India, 2007.

[44] C. Karfa, C. Mandal, and D. Sarkar. Formal verification of code motion techniques using data-flow-driven equivalence checking. *ACM Trans. Design Autom. Electr. Syst.*, 17(3):30:1–30:37, 2012.

[45] C. Karfa, C. Mandal, D. Sarkar, S. R. Pentakota, and C. Reade. A formal verification method of scheduling in high-level synthesis. In *ISQED*, pages 71–78, 2006.

[46] C. Karfa, D. Sarkar, and C. Mandal. Verification of datapath and controller generation phase in high-level synthesis of digital circuits. *IEEE Trans. on CAD of ICS*, 29(3):479–492, 2010.

[47] C. Karfa, D. Sarkar, C. Mandal, and P. Kumar. An equivalence-checking method for scheduling verification in high-level synthesis. *IEEE Trans on CAD of ICS*, 27:556–569, 2008.

[48] Y. Kim, S. Kopuri, and N. Mansouri. Automated formal verification of scheduling process using finite state machines with datapath (fsmd). In *Quality Electronic Design, 2004. Proceedings. 5th International Symposium on*, pages 110 – 115, 2004.

[49] J. C. King. *A program verifier*. PhD thesis, Pittsburgh, PA, USA, 1970.

[50] A. Kolawa and D. Huizinga. *Automated Defect Prevention: Best Practices in Software Management*. Wiley-IEEE Computer Society Press, 2007.

[51] A. Komuravelli, A. Gurfinkel, and S. Chaki. Smt-based model checking for recursive programs. In *Proceedings of the 16th International Conference on Computer Aided Verification - Volume 8559*, pages 17–34, New York, NY, USA, 2014. Springer-Verlag New York, Inc.

[52] R. Könighofer and R. Bloem. Automated error localization and correction for imperative programs. In *Proceedings of the International Conference on Formal Methods in Computer-Aided Design*, FMCAD '11, pages 91–100, Austin, TX, 2011. FMCAD Inc.

[53] J. P. Leal and F. Silva. Mooshak: a web-based multi-site programming contest system. *Software: Practice and Experience*, 33(6):567–581, 2003.

[54] C.-H. Lee, C.-H. Shih, J.-D. Huang, and J.-Y. Jou. Equivalence checking of scheduling with speculative code transformations in high-level synthesis. In *ASP-DAC*, pages 497–502, 2011.

[55] Y. Liang, Q. Liu, J. Xu, and D. Wang. The recent development of automated programming assessment. In *2009 International Conference on Computational Intelligence and Software Engineering*, pages 1–5, Dec 2009.

[56] N. P. Lopes and J. Monteiro. Automatic equivalence checking of UF+IA programs. In *SPIN*, pages 282–300, 2013.

[57] L. Malmi, A. Korhonen, and R. Saikkonen. Experiences in automatic assessment on mass courses and issues for designing virtual courses. *SIGCSE Bull.*, 34(3):55–59, June 2002.

[58] A. K. Mandal, C. Mandal, and C. Reade. A system for automatic evaluation of programs for correctness and performance. In J. A. M. Cordeiro, V. Pedrosa, B. Encarnação, and J. Filipe, editors, *WEBIST (2)*, pages 196–203. INSTICC Press, 2006.

[59] Z. Manna. *Mathematical Theory of Computation*. McGraw-Hill, New York, 1974.

[60] T. Matsumoto, H. Saito, and M. Fujita. Equivalence checking of c programs by locally performing symbolic simulation on dependence graphs. In *Quality Electronic Design, 2006. ISQED '06. 7th International Symposium on*, pages 6 pp. –375, march 2006.

[61] F. Merz, S. Falke, and C. Sinz. LLBMC: Bounded model checking of `C` and `C++` programs using a compiler IR. In *Proceedings of the 4th International Conference on Verified Software: Theories, Tools, Experiments*, VSTTE'12, pages 146–161, Berlin, Heidelberg, 2012. Springer-Verlag.

[62] M. Mitzenmacher and E. Upfal. *Probability and Computing: Randomized Algorithms and Probabilistic Analysis*. Cambridge University Press, New York, NY, USA, 2005.

[63] D. S. Morris. Automatic grading of student's programming assignments: an interactive process and suite of programs. In *33rd Annual Frontiers in Education, 2003. FIE 2003.*, volume 3, pages S3F–1–6 vol.3, Nov 2003.

[64] K. A. Naudé, J. H. Greyling, and D. Vogts. Marking student programs using graph similarity. *Computers & Education*, 54(2):545 – 561, 2010.

[65] V. Pieterse. Automated assessment of programming assignments. In *Proceedings of the 3rd Computer Science Education Research Conference on Computer Science Education Research*, CSERC '13, pages 4:45–4:56, Open Univ., Heerlen, The Netherlands, The Netherlands, 2013. Open Universiteit, Heerlen.

[66] D. Qi, A. Roychoudhury, Z. Liang, and K. Vaswani. Darwin: An approach to debugging evolving programs. *ACM Trans. Softw. Eng. Methodol.*, 21(3):19:1–19:29, July 2012.

[67] K. Rahman, M. Nordin, and W. Che. Automated programming assessment using the pseudocode comparison technique: Does it really work? In *Information Technology, 2008. ITSim 2008. International Symposium on*, volume 3, pages 1 –4, Aug. 2008.

[68] K. A. Rahman, M. J. Nordin, and S. Ahmad. The design of an automated c programming assessment using pseudo-code comparison technique. In *paper read at National Conference on Software Engineering and Computer Systems, at University Malaysia Pahang,Malaysia, 2007.*, pages 1 –10. (available from http://myais.fsktm.um.edu.my/1788/).

[69] K. A. Reek. The try system -or- how to avoid testing student programs. In *Proceedings of the Twentieth SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '89, pages 112–116, New York, NY, USA, 1989. ACM.

[70] H. Rocha, H. Ismail, L. C. Cordeiro, and R. S. Barreto. Model checking `C` programs with loops via k-induction and invariants. *CoRR*, abs/1502.02327, 2015.

[71] G. Roy. Techniques and algorithms for the design and development of a virtual laboratory to support logic design and computer organization. Master's thesis, IIT Kharagpur, India, 2014.

[72] M. Rubio-Sánchez, P. Kinnunen, C. Pareja-Flores, and Á. Velázquez-Iturbide. Student perception and usage of an automated programming assessment tool. *Computers in Human Behavior*, 31:453 – 460, 2014.

[73] P. Rudnicki. Little bezout theorem (factor theorem). *FORMALIZED MATHEMATICS*, 12:2004, 2004.

[74] R. Saikkonen, L. Malmi, and A. Korhonen. Fully automatic assessment of programming exercises. In *Proceedings of the 6th Annual Conference on Innovation and Technology in Computer Science Education*, ITiCSE '01, pages 133–136, New York, NY, USA, 2001. ACM.

[75] D. Sarkar and S. C. De Sarkar. A theorem prover for verifying iterative programs over integers. *IEEE Trans Software. Engg.*, 15(12):1550–1566, 1989.

[76] F. Shamsi and A. Elnagar. An intelligent assessment tool for students' java submissions in introductory programming courses. *Intelligent Learning Systems and Applications*, 4(1):59–69, 2012.

[77] K. K. Sharma, K. Banerjee, and C. Mandal. A scheme for automated evaluation of programming assignments using FSMD based equivalence checking. In *I-CARE*, pages 10:1–10:4, 2014.

[78] K. K. Sharma, K. Banerjee, and C. Mandal. Determining equivalence of expressions: An automated evaluator's perspective. In *Technology for Education (T4E), 2015 IEEE International Conference on*, pages 35–36, 2015.

[79] K. K. Sharma, K. Banerjee, and C. Mandal. Establishing equivalence of expressions: An automated evaluator designer's perspective. In *Mining Intelligence and Knowledge Exploration - Third International Conference, MIKE 2015, Hyderabad, India, December 9-11, 2015, Proceedings*, pages 415–423, 2015.

[80] K. K. Sharma, K. Banerjee, C. Mandal, and I. Vikas. A benchmark programming assignment suite for quantitative analysis of student performance in early programming courses. In *MOOC, Innovation and Technology in Education (MITE), 2015 IEEE International Conference on*, pages 199–203, 2015.

[81] K. K. Sharma, K. Banerjee, I. Vikas, and C. Mandal. Automated checking of the violation of precedence of conditions in else-if constructs in students' programs. In *MOOC, Innovation and Technology in Education (MITE), 2014 IEEE International Conference on*, pages 201–204, 2014.

[82] G. Simons. Lower bounds for average sample number of sequential multihypothesis tests. *Ann. Math. Statist.*, 38(5):1343–1364, 10 1967.

[83] R. Singh, S. Gulwani, and A. Solar-Lezama. Automated semantic grading of programs. *CoRR*, abs/1204.1751, 2012.

[84] M. Sipser. *Introduction to the Theory of Computation*. International Thomson Publishing, 1st edition, 1996.

[85] S. Subramanian, M. Berzish, Y. Zheng, O. Tripp, and V. Ganesh. A solver for a theory of strings and bit-vectors. *CoRR*, abs/1605.09446, 2016.

[86] G. Tremblay, F. Guérin, A. Pons, and A. Salah. Oto, a generic and extensible tool for marking programming assignments. *Softw. Pract. Exper.*, 38(3):307–333, Mar. 2008.

[87] J.-B. Tristan and X. Leroy. Verified validation of lazy code motion. *SIGPLAN Not.*, 44(6):316–326, June 2009.

[88] M. Vujošević-Janičić and V. Kuncak. *Development and Evaluation of LAV: An SMT-Based Error Finding Platform*, pages 98–113. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[89] M. Vujošević-Janičić, M. Nikolić, D. Tošić, and V. Kuncak. Software verification and graph similarity for automated evaluation of students' assignments. *Information and Software Technology*, 55(6):1004 – 1016, 2013.

[90] T. Wang, X. Su, P. Ma, Y. Wang, and K. Wang. Ability-training-oriented automated assessment in introductory programming course. *Computers & Education*, 56(1):220 – 226, 2011.

[91] T. Wang, X. Su, Y. Wang, and P. Ma. Semantic similarity-based grading of student programs. *Info. and Software Technology*, 49(2):99 – 107, 2007.

[92] O. Watanabe. Sequential sampling techniques for algorithmic learning theory. *Theoretical Computer Science*, 348(1):3 – 14, 2005. Algorithmic Learning Theory (ALT 2000).

[93] S. Xu and Y. S. Chee. Transformation-based diagnosis of student programs for programming tutoring systems. *Software Engineering, IEEE Transactions on*, 29(4):360 – 384, April 2003.

[94] Y. Yu, Z. Duan, C. Tian, and M. Yang. Model checking `C` programs with MSVL. In S. Liu, editor, *Structured Object-Oriented Formal Language and Method*, pages 87–103, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[95] Z. Zhou and W. Burleson. Equivalence checking of datapaths based on canonical arithmetic expressions. In *32nd Design Automation Conference*, pages 546–551, 1995.

[96] A. M. Zin, S. A. Aljunid, Z. Shukur, and M. J. Nordin. A knowledge-based automated debugger in learning system. In *AADEBUG*, 2000.

Publications

1. Sharma, K. K., Banerjee, K., Mandal, C.: A scheme for automated evaluation of programming assignments using FSMD based equivalence checking. In: I-CARE, pp. 10:1–10:4 (2014). ACM Digital Library

2. Sharma, K. K., Banerjee, K., Vikas, I., Mandal, C.: Automated checking of the violation of precedence of conditions in else-if constructs in students' programs. In: 2014 IEEE International Conference on MOOC, Innovation and Technology in Education (MITE), pp. 201–204 (2014), ieeexplore.ieee.org

3. Sharma, K. K., Banerjee, K., Mandal, C.: Determining equivalence of expressions: an automated evaluator's perspective. In: 2015 IEEE International Conference on Technology for Education (T4E) (2015), pp. 35–36, ieeexplore.ieee.org

4. Sharma, K. K., Kunal Banerjee, and Chittaranjan Mandal. Establishing Equivalence of Expressions: An Automated Evaluator Designer's Perspective. In: International Conference on Mining Intelligence and Knowledge Exploration (MIKE), pp. 415–423. Springer International Publishing, (2015).

5. Sharma, K. K., Banerjee, K., Mandal, C., Vikas, I.: A benchmark programming assignment suite for quantitative analysis of student performance in early programming courses. In: 2015 IEEE International Conference on MOOC, Innovation and Technology in Education (MITE) (2015), pp. 199–203, ieeexplore.ieee.org

About the author

K. K. Sharma is a research scholar in CSE Deptt. of IIT Kharagpur under QIP scheme of Govt. of India. He holds a B.E. (Elecronics) from MNIT Jaipur, India and M.E. (Computer Engg.) from SGSITS Indore, India. He has been working as Associate Professor in the Information Technology Department at SGSITS Indore for about a decade.