FORMAL VERIFICATION OF BEHAVIOURAL TRANSFORMATIONS DURING EMBEDDED SYSTEM DESIGN

Chandan Karfa

FORMAL VERIFICATION OF BEHAVIOURAL TRANSFORMATIONS DURING EMBEDDED SYSTEM DESIGN

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

of

Doctor of Philosophy

by

Chandan Karfa

Under the supervision of

Dr. Chittaranjan Mandal
and
Dr. Dipankar Sarkar

Department of Computer Science and Engineering

Indian Institute of Technology, Kharagpur

September 2011

APPROVAL OF THE VIVA-VOCE BOARD

Certified that the thesis entitled **“Formal Verification of Behavioural Transformations during Embedded System Design"** submitted by **Chandan Karfa** 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.

(Member of the DSC)	(Member of the DSC)
(Member of the DSC)	(Member of the DSC)
(Supervisor)	(Supervisor)
(External Examiner)	(Chairman)

Date:

CERTIFICATE

This is to certify that the thesis entitled **"Formal Verification of Behavioural Transformations during Embedded System Design"**, submitted by **Chandan Karfa** 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.

Chittaranjan Mandal	Dipankar Sarkar
Professor	Professor
CSE, IIT Kharagpur	CSE, IIT Kharagpur

Date:

DECLARATION

I certify that

(a) The work contained in the 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.

Chandan Karfa

ACKNOWLEDGMENTS

I would like to express my profound sense of gratitude to my thesis supervisors Prof. Chittaranjan Mandal and Prof. Dipankar Sarkar for guiding me through the PhD program. I acknowledge their constant technical and moral support and guidance throughout my PhD. I have learned a lot about the art of conducting research and solving problems from my supervisors. I consider myself extremely lucky for getting the opportunity to work under them.

I acknowledge the support from Microsoft Research India for partially funding my research work and providing generous travel grants and stipend through the Microsoft Research India PhD Fellowship Award.

I want to thank my labmates Arnab da, Aritra, Priyankar da, Srobona, MSD, Kunal, Partha, Antara, Debjit, Sudip, Subho, Gautam, Debi da and others for making the lab most enjoyable place in Kgp. I want to thank Debjit for all his system and latex related supports. Special thanks to Kunal and Partha for helping me in some experiments of my work. My stay in Kgp was extremely pleasurable thanks to my friends Somnath, Soumyajit, Soumyadip, Bodhi, Pradipta da, Gopal da, Vivek, Sumit, Sayan, Soubhik, Rajarshi and many others. I spent a lot of good time in TATA sports complex with VSRC football and cricket teams.

I acknowledge google.com for helping me in searching most of the documents during my research. Special thanks to cricinfo.com which is the source of my another (re)search on cricket during my stay in Kgp.

Last but most important are my wife Aparna, my parents and the other members of my family. Without their constant encouragements, supports, love and well wishes, this thesis never be materialized.

Chandan Karfa

ABSTRACT

Application of behavioural transformations for obtaining optimal performance, energy and/or area on a given platform during embedded system design is now a common practice. Verifying correctness of these transformations is an important step in ensuring dependability of embedded systems. This thesis addresses verification methodologies, primarily by way of equivalence checking, for six behavioural transformations that are applied during embedded system design. The transformations considered cover code motion, generation of register transfer level (RTL) design after carrying high-level optimizations and also RTL transformations, loop and arithmetic transformations on array based programs, transformations on array based programs leading to the generations of Kahn process networks (KPN) to achieve high degree of parallelism and also transformations applied at the KPN level.

Verification methods for the first three transformations on programs not involving arrays employ the model of finite state machines with datapaths (FSMD). FSMDs are generalizations of FSMs to capture data transformations taking place between control states. FSMD based equivalence checking method consists in introducing cutpoints in one FSMD, visualizing its computations as concatenation of paths from cutpoints to cutpoints and finally, identifying equivalent finite path segments in the other FSMD; the process is then repeated with the FSMDs interchanged. Verification methods for the last three transformations involving arrays employ the array data dependence graphs (ADDG). The ADDG model primarily captures computation of a range of elements of an array from ranges of elements of other arrays. ADDG based equivalence checking attempts to show that the overall computations depicted in the ADDG to define the final ADDG nodes in terms of the initial ADDG nodes are equivalent, both in terms of computation and range of array elements, with respect to those in the ADDG being compared.

The FSMD based methods developed handle both uniform and non-uniform code motion transformations. For non-uniform code motions model checking of some data-flow properties is also carried out. A rewriting mechanism covering both pipelining and multicycling is used to construct an FSMD from register transfer operations for RTL related equivalence checking. In our ADDG equivalence checking method we have introduced the notion of slices to guide the checking along the sequences in which operations are used to define array elements in the original behaviour. Normalized representation of arithmetic and conditional expressions play a central role in handling arithmetic transformations for both FSMD and ADDG based equivalence checking. Our ADDG based checking has been extended to check the equivalence of KPN networks derived by parallelizing sequential array based programs. Potential deadlocks in the KPN are also detected through the ADDG based modelling.

Correctness and complexity of the all the developed methods have been treated formally. The methods have been implemented and tested on several benchmarks. This work represents useful application of equivalence checking as a verification method for verification to several aspects of the embedded system design flow.

Keywords: Embedded System, Formal Verification, Equivalence Checking, Behavioural Transformation, Code Motion, Loop Transformation, Register Transfer Level (RTL), Kahn Process Network, Finite State Machine with Datapath (FSMD), Array Data Dependence Graph (ADDG).

Contents

Abstract **xiii**

Table of Contents **xv**

List of Symbols **xix**

List of Figures **xxi**

List of Tables **xxv**

1 Introduction **1**

1.1 Embedded system design flow 1

1.2 Behavioural transformations 4

1.2.1 Code motion transformations 4

1.2.2 Loop transformations 5

1.2.3 Arithmetic transformations 6

1.2.4 High-level to RTL transformations 6

1.2.5 Sequential to parallel transformations 7

1.3 Motivations and objectives 8

1.3.1 Problem statements 10

1.4 Contributions 13

1.5 Organization of the thesis 15

2 Literature Survey **17**

2.1 Code motion transformations 17

2.1.1 Applications of code motion transformations 17

2.1.2 Verification of code motion transformations 20

2.2 High-level to RTL and RTL transformations 22

2.2.1 Applications of High-level to RTL and RTL transformations . 22

2.2.2 Verification of High-level to RTL and RTL transformations . . 23

2.3 Loop transformations and arithmetic transformations 25
2.3.1 Applications of loop transformations 25
2.3.2 Applications of arithmetic transformations 27
2.3.3 Verification of loop and arithmetic transformations 29
2.4 Parallelizing transformations 31
2.4.1 Applications of parallelizing transformations 31
2.4.2 Verification of parallelizing transformations 34
2.5 Conclusion 35

3 Verification of Code Motion Transformations 37
3.1 Introduction 37
3.2 Basic equivalence checking method 38
3.2.1 FSMDs and its paths 38
3.2.2 Normalization of arithmetic expressions 42
3.3 Equivalence problem formulation 44
3.3.1 Path cover and equivalence of FSMDs 45
3.3.2 A method to handle uniform code motions 46
3.4 Verification of non-uniform code motions 48
3.4.1 An example of non-uniform code motion 48
3.4.2 A scheme for verifying non-uniform code motions 49
3.4.3 Strong and weak equivalence of paths 53
3.4.4 Formulation of the path extension procedure 56
3.4.5 Encoding and model checking the data-flow properties 57
3.4.6 The equivalence checking method 58
3.4.7 Illustration of working of the equivalence checking method . . 64
3.4.8 Justification of the initial cutpoints 66
3.5 Multicycle and pipelined execution of operations 68
3.6 Correctness and complexity 70
3.6.1 Correctness 70
3.6.2 Complexity 74
3.7 Experimental results 75
3.7.1 Limitations of the method 81
3.8 Conclusion 82

4 Verification of RTL Generation Phase 83
4.1 Introduction 83
4.2 Verification challenges 86
4.3 Construction of FSMDs from RTL designs 87
4.3.1 Representation of the datapath description 87
4.3.2 A Method of obtaining the micro-operations for a control assertion pattern 89
4.3.3 Identification of RT operations realized by a set of micro-operations 91
4.3.4 Multicycle, pipelined and chained operations 93
4.4 The Overall construction framework of FSMD 95
4.4.1 Handling of multicycle operations 96

4.4.2 Handling of pipelined operations . . . 99
4.4.3 Handling chained operations . . . 101
4.4.4 Verification during construction of FSMD . . . 102
4.5 Correctness and complexity of the algorithm . . . 103
4.5.1 Correctness and complexity of the module *findRewriteSeq* . . 103
4.5.2 Correctness and complexity of the module *RTLV-1* . . . 111
4.5.3 Correctness and complexity of the modules *Multicycle* and *Pipelined* . . . 112
4.5.4 Correctness and complexity of the module *RTLV-0* . . . 113
4.6 Verification by equivalence checking . . . 116
4.7 Verification of low power RTL transformations . . . 117
4.7.1 Alternative datapath architecture . . . 118
4.7.2 Restructuring of multiplexer networks to enhance data correlation . . . 121
4.7.3 Restructuring of multiplexer networks to eliminate glitchy control signals . . . 122
4.7.4 Clocking of control signals . . . 123
4.7.5 Glitch reduction using delays . . . 124
4.8 Experimental results . . . 126
4.9 Conclusion . . . 132

5 Verification of Loop Transformations **133**
5.1 Introduction . . . 133
5.2 Array data dependence graphs . . . 135
5.2.1 Representation of data dependencies of the behaviour . . . 139
5.2.2 Transitive dependence . . . 142
5.2.3 Recurrence in ADDG . . . 146
5.2.4 Construction of the ADDG from a sequential behaviour . . . 148
5.3 Slices . . . 148
5.4 Equivalence of ADDGs . . . 154
5.4.1 Normalization of the characteristic formula of a slice . . . 156
5.4.2 Some simplification rules for data transformations . . . 158
5.4.3 Equivalence problem formulation . . . 161
5.5 A case study . . . 165
5.6 Correctness and complexity . . . 167
5.6.1 Complexity . . . 168
5.7 Error diagnosis . . . 171
5.8 Experimental results . . . 172
5.9 Conclusion . . . 174

6 Verification of Parallelizing Transformations **175**
6.1 Introduction . . . 175
6.2 Verification framework . . . 176
6.2.1 Kahn process networks . . . 176
6.2.2 Verification approach . . . 176
6.3 Modelling a KPN as an ADDG . . . 179

6.3.1 Modelling KPN processes as ADDGs 179
6.3.2 Computation of the dependence mappings involving FIFOs . . 182
6.3.3 Composition of ADDGs of KPN processes 191
6.3.4 Correctness of the composition operation 195
6.4 Deadlock detection in a KPN 199
6.4.1 Deadlock due to insufficient communication 200
6.4.2 Deadlock due to circular dependence in a KPN 201
6.5 Verification of KPN level transformations 205
6.5.1 Channel merging 205
6.5.2 Channel splitting 213
6.5.3 Process splitting 217
6.5.4 Process merging 223
6.5.5 Message vectorization 226
6.5.6 Computation migration 230
6.6 Experimental results 233
6.7 Conclusion 234

7 Conclusion and Future Scopes 237
7.1 Summary of contributions 238
7.2 Scope for future work 242
7.2.1 Enhancement of the present work 242
7.2.2 Scope of application to other research areas 242
7.3 Conclusion 244

Bibliography 249

List of Symbols

M_0, M_1	FSMDs	39
$\mathcal{S}$	Set of status expressions	39
V_0, V_1	Set of storage variables of FSMDs M_0 and M_1	38
B	A set of Boolean variables	39
Q_0, Q_1	Set of states of FSMD M_0 and M_1	39
$q_{0,i}$	A state of FSMD M_0	44
$q_{1,i}$	A state of FSMD M_1	44
f	State transition function of FSMD	39
u	Update function of FSMD	39
α, β	Paths in FSMD	40
R_α	Condition of execution of the path α	40
r_α	Data transformation of the path α	41
s_α	Storage variable transformations of the path α	41
$s_\alpha\|_{V'}$	Storage variable transformations restricted over the variable set V'	54
O_α	Output list of the path α	41
$\mathsf{endPtNd}(\beta)$	End state of the path β	56
τ_α	Characteristic formula of the path α	42
S	Normalized sum	43
T	Term in normalized sum	43
P	Primary in terms of a normalized sum	43
R	Set of relational operators	39
c, c_1, c_2	Computations in FSMDs	45
$c_1 \simeq c_2$	Equivalence of computations c_1 and c_2	45
$M_0 \sqsubseteq M_1$	An FSMD M_0 is contained in an FSMD M_1	45
P_0, P_1	Path cover of the FSMDs M_0 and M_1	46
$\alpha \simeq_s \beta$	Strong equivalence of paths α and β	54
$\alpha \simeq_w \beta$	Weak equivalence of paths α and β	54
$\langle q_{0,i},\ q_{1,j} \rangle$	$q_{0,i}$ and $q_{1,j}$ are corresponding state pair	61
ζ	The set of corresponding state pairs	61
$\mathbb{N}$	The set of natural numbers	68
$\mathscr{M}$	The set of all possible micro-operations in the datapath	87
$\mathscr{A}$	The set of all possible control assertion patterns	88
μ	A micro-operation	88

ρ	A control assertion pattern	88
f_{mc}	A relation $\subseteq \mathscr{M} \times \mathscr{A}$ such that $(\mu, \rho) \in f_{mc}$ when ρ is needed for μ	88
$\mathscr{M}_A$	The set of activated by the control assertion pattern A	89
$A_1 \theta A_2$	Superposition of assertion patterns A_1 and A_2	89
$\pi_i(A)$	The *i*-th projection of an assertion pattern A	89
σ	A sequence of micro-operations	104
G, G_S, G_T	ADDGs	135
A	The set of array nodes in ADDG	135
F	The set of operator nodes in ADDG	135
I_S	Iteration domain of the statement S	139
$\vec{v}$	An iteration vector	140
${}_S M_d^{(d)}$	Definition mapping	140
${}_S D_d$	Definition domain	140
${}_S M_{u_n}^{(u)}$	Operand mapping	141
${}_S U_{u_n}$	Operand domain	141
${}_S M_{d,u_n}$	Dependence mapping	141
$\diamond$	Right composition operator	143
g, g_1, g_2	Slices in ADDG	149
r_g	Data transformation of a slice g	151
τ_g	Characteristic formula of a slice g	151
L_I	Left-hand-side (defined) array index expressions	157
R_I	Right-hand-side (used) array index expressions	157
D_Q	Quantified formula depicting the domain of the variables in L_I and R_I	157
$g_1 \approx g_2$	g_1 and g_2 are matching IO-slices	161
$\mathcal{C}_{G_S}$	IO-slice class of the ADDG G_S	164
$C_1 \simeq C_2$	The IO-slice classes C_1 and C_2 are equivalent	162
$\emptyset$	The empty set	210

List of Figures

1.1 Embedded system design flow 2
1.2 HW-SW partitioning 3
1.3 Various code motion techniques 4

2.1 A program with non-uniform recurrence on array A 30
2.2 A program with data dependent control and array access 30
2.3 Commutative and distributive transformations: (a) original program; (b) transformed program 30

3.1 Example of reverse speculation: (a) and (c): Two FSMDs are shown combined representing two behaviours before scheduling; (b) and (d): Two FSMDs are shown combined representing the behaviours after scheduling of the FSMD in (a) and in (c), respectively 40
3.2 An example of duplicating down 47
3.3 An instance of non-uniform code motion 50
3.4 An example of speculative code motion 51
3.5 Kripke structure obtained from FSMD M_0 of figure 3.1(a) 57
3.6 An example of cutpoint selection 67
3.7 (a) A 2-cycle multiplier; (b) A 3-stage pipelined multiplier; (c) A sample path in an FSMD with multicycle operation; (d) A sample path in an FSMD with pipelined operation 69
3.8 Illustration of deletion operation 71

4.1 Hand-in-hand synthesis and verification framework 84
4.2 The steps of datapath and controller verification 85
4.3 Scheduling of a relational operation 86
4.4 Datapath with control signals 88
4.5 (a). schedule with a 2-cycle multiplier; (b) schedule with a 3-stage pipelined multiplier; (c) schedule with a chained adder and subtracter; (d) Input and output timing of a *k*-cycle multiplier; (e) Input and output timing of a *k*-stage pipelined multiplier; (f) Input and output timing for a chained adder and subtracter 94
4.6 Schematic of our FSMD construction framework from the datapath description and the controller FSM 95

4.7 Working with multicyle and pipelined operations 100
4.8 Parse tree of an RT-operation realized over a given datapath: An Example 106
4.9 Verification framework 117
4.10 Alternative datapath architecture 119
4.11 Multiplexer network restructuring to enhance data correlation: An example 121
4.12 Multiplexer network restructuring to eliminate glitchy control signals: An example 123
4.13 Clocking control signal to kill glitch 124
4.14 Insertion of delay element in signal line 125
4.15 Insertion of delay element in data inputs 126

5.1 The ADDG of the sequential behaviour given in example 10 137
5.2 A generalized *x*-nested loop structure 138
5.3 Computation of dependence mapping 141
5.4 Transitive dependence 143
5.5 (a) A program with recurrence; (b) the ADDG of the program 146
5.6 Slices of the ADDG in figure 5.1 149
5.7 (a) A program, (b) its ADDG and (c)-(d) two slices of the ADDG . . . 149
5.8 An example of computation of data transformation over a slice 150
5.9 (a) source program; (b) transformed program 155
5.10 (a) ADDG of the source program; (b) ADDG of the transformed program 156
5.11 An example for matching slices and slice class 162
5.12 (a) original behaviour; (b) transformed behaviour 165
5.13 (a) ADDG of the original program; (b) ADDG of the transformed program 166

6.1 Verification of parallelizing transformations by equivalence checking . 177
6.2 The KPN representation of the concurrent behaviour 179
6.3 Behaviours of the processes of the KPN given in figure 6.2 180
6.4 Modified behaviours of the processes given in figure 6.3 181
6.5 The ADDGs of processes of the KPN behaviour in figure 6.2 182
6.6 A generalized nested loop behaviour 183
6.7 An example of nonlinear relation between $f1In$ and loop indices (i, j) 185
6.8 An process of a KPN 186
6.9 An example for under constrained linear system of equations 187
6.10 A FIFO is written by more than one statement 190
6.11 Composition of ADDGs 193
6.12 (a) The ADDG of the sequential behaviour given in example 21; (b) The ADDG of the corresponding KPN behaviour 194
6.13 An example of producer-consumer deadlock 197
6.14 A KPN with circular dependence 201
6.15 ADDGs of the processes in figure 6.14: (a) ADDG of process P1; (b) ADDG of process P2; (c) Composed ADDG of P1 and P2 202
6.16 Another KPN of the with circular dependence: (a) Processes of the KPN; (b) ADDG of the KPN 203

6.17 Channel Merging: (a) structure of the original KPN and (b) structure of transformed KPN 206
6.18 Channel Merging: (a) behaviours of the processes in initial KPN with two processes and two FIFOs; (b) behaviours of the processes after merging two channels; (c) Erroneous behaviour of process P_2 207
6.19 Channel Merging: (a) Composed ADDG of the initial KPN; (b) Composed ADDG of the transformed KPN; (c) Composed ADDG of the transformed KPN when P_2 is as in figure 6.18(c) 208
6.20 Channel Splitting: (a) initial KPN; (b) the transformed KPN 213
6.21 Channel Splitting: (a) A KPN with two processes and one channels; (b) The KPN after splitting the channel into two 214
6.22 Channel Splitting: (a) Composed ADDG of the input KPN; (b) Composed ADDG of the transformed KPN 215
6.23 Process Splitting: (a) initial KPN; (b) transformed KPN after process splitting 218
6.24 Process Splitting: (a) A KPN with two processes and one channel; (b) The KPN after splitting the process P_2 into two processes P_{2a} and P_{2b} 219
6.25 Process Splitting: (a) ADDG of the input KPN and (b) ADDG of the transformed KPN 220
6.26 The KPN after splitting process P_1 of KPN in figure 6.21; (b) into two process P_{1a} and P_{1b} 222
6.27 Process Merging: (a) the initial KPN; (b) the transformed KPN 223
6.28 Process Merging: A KPN with four processes and four channels; (b) The KPN comprising P_1, P_{2a} and P_4 obtained after merging two processes P_2 and P_3 (into P_{2a}) 224
6.29 Process Merging: (a) ADDG of the input KPN; (b) ADDG of the transformed KPN; (c) ADDG of the KPN in figure 6.30 225
6.30 KPN after merging channels FIFO1 and FIFO2 into FIFO2 and channels FIFO3 and FIFO4 into FIFO3 226
6.31 Message Vectorization: (a) initial KPN with two processes and one channel; (b) The KPN when message is sent as a vector of 10 elements 227
6.32 Message de-vectorization: Modified behaviour of the KPN in figure 6.31(b) 227
6.33 Message de-vectorization: (a) ADDG of the input KPN; (b) ADDG of the transformed KPN 228
6.34 Computation Migration: (a) the structure of the initial KPN; (b) the structure of the KPN after computation migrations 231
6.35 Computation Migration: (a) A KPN with three processes and four channels; (b) The KPN after migration of computation f_3 from process P2 to P1 232
6.36 Computation Migration: (a) ADDG of the initial KPN; (b) ADDG of the transformed KPN 233

List of Tables

3.1 Characteristics of the HLS benchmarks and synthesis results of SPARK 76
3.2 Results for several high-level synthesis benchmarks 77
3.3 Results for verification of uniform code motions 78
3.4 Comparison time with competitive methods for uniform code motions 78
3.5 Results for several high-level synthesis benchmarks on erroneous design 80

4.1 Construction of the set $\mathscr{M}_A$ from f_{mc} for the control assertion pattern $A = \langle 1,0,1,0,1,0,1,1,0 \rangle$ 90
4.2 The micro-operations of datapath in figure 4.10(b) and computation of $\mathcal{M}_{\mathcal{A}}$ for CAP $\langle 0,1 \rangle$ 120
4.3 The micro-operations of datapath in figure 4.10(b) and computation of $\mathcal{M}_{\mathcal{A}}$ for CAP $\langle 0,1 \rangle$ 120
4.4 Characteristics of the HLS benchmarks used in our experiment 127
4.5 Results for several high-level synthesis benchmarks 128
4.6 Results for multicycle and pipelined datapath for two high-level synthesis benchmarks 129

5.1 Results for several benchmarks 172

6.1 Results for sequential to KPN transformation 234
6.2 Results for KPN level transformations 234

Chapter 1

Introduction

Embedded systems (ES) are information processing systems that are embedded into different products. These systems are dedicated towards certain applications with real-time constraints, reactive in nature, and must be dependable and efficient (Marwedel, 2006). Embedded systems pervade all aspects of modern life, spanning from consumer electronics, household appliances, automotive electronics, aircraft electronics and telecommunication systems. These systems are also suited for use in transportation, safety and security, robotics, medical applications and life critical systems. In the subsequent sections, we describe the ES design flow and highlight various behavioural transportation that are often applied in course of the design. Since dependability is an important aspect of ES, we have taken up the issue of verifying the correctness of such transformations by way of equivalence checking. Specific problems handled in the thesis have been listed. This is followed by a statement of contributions and the thesis organization.

1.1 Embedded system design flow

Present day electronic products demand high performance and extensive features. As a consequence, embedded systems are becoming more complex and difficult to design. To combat complexity and explore the design space effectively, it is necessary to represent systems at multiple levels of abstraction (Chen et al., 2006b). Initial functions and architectures are preferably specified at a high level of abstraction. Functions

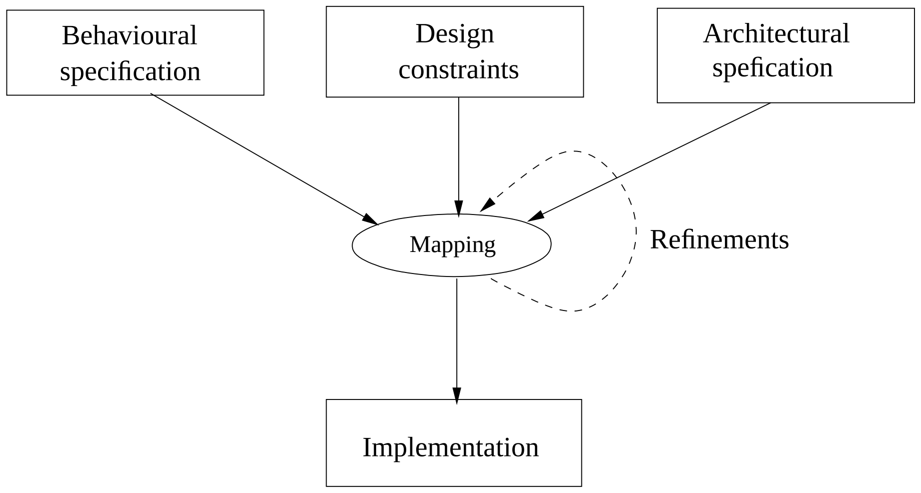


Figure 1.1: Embedded system design flow

are then mapped onto the architecture through an iterative refinement process (Chen et al., 2003; Keutzer et al., 2000). The embedded system design flow based on this principle is shown in figure 1.1. During the refinement process, the initial design is repeatedly partitioned into hardware (HW) and software (SW) parts – the software part executes on (multi)processor systems and the hardware part runs as circuits on some IC fabric like an ASIC or FPGA. Application-specific hardware is usually much faster than software, but its design, validation and fabrication are significantly more expensive. Software, on the other hand, is cheaper to create and to maintain, but it is slow. Therefore, the performance-critical components of the system should be realized in hardware, and non-critical components should be realized in software. The HW-SW partitioning approach is shown in figure 1.2.

The SW part of the behaviour is translated into assembly/machine language code by compilers. The translation process of input behaviour into target assembly language code can be divided into two stages: the front end for analysis and the back end for synthesis (Raghavan, 2010). The front end of the compiler transforms the input behaviour into an intermediate representation (IR) and then the back end transforms the IRs into a target assembly language code. The front end consists of several subtasks such as lexical analysis, syntax analysis, semantic analysis, intermediate code generation and optimizations (Aho et al., 1987; Raghavan, 2010). The back end of the compiler consists of target code generation and optimization tasks. A set of behavioural transformations are applied in the optimization phase by modern compilers during this translation processes (Muchnick, 1997). Figure 1.2 highlights the list of

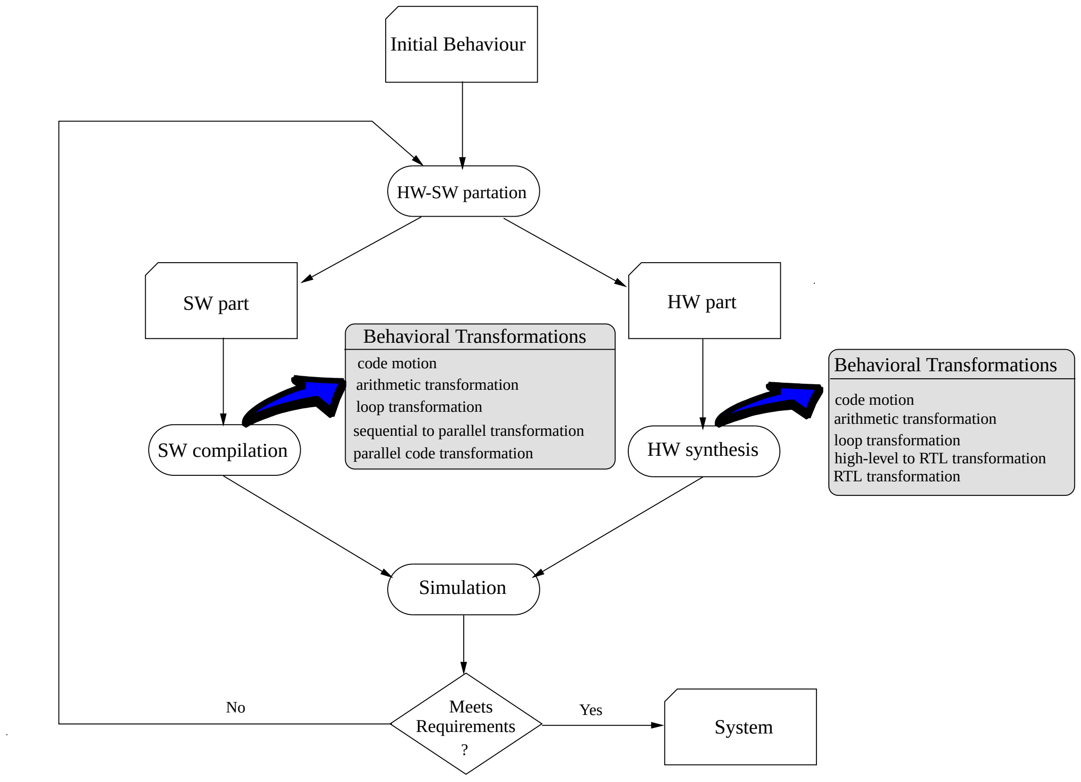

Initial Behaviour
HW-SW partation
SW part
HW part
Behavioral Transformations
code motion
arithmetic transformation
loop transformation
sequential to parallel transformation
parallel code transformation
SW compilation
HW synthesis
Behavioral Transformations
code motion
arithmetic transformation
loop transformation
high-level to RTL transformation
RTL transformation
Simulation
No
Meets Requirements ?
Yes
System

Figure 1.2: HW-SW partitioning

behavioural transformations that are applied during SW-compilation processes.

The HW part of the behaviour is first translated into a register transfer level (RTL) code by high level synthesis (HLS) tools (Gajski et al., 1992; Gupta et al., 2003c). The generated RTL then goes through the logic synthesis and the physical design process before being fabricated into a chip. The HLS process consists of several interdependent subtasks such as compilation or pre-processing, scheduling, allocation and binding, and datapath and controller generation. The HLS process applies several behavioural transformations during pre-processing and scheduling phases. The list of behavioural transformations that are commonly used during these phases are listed in figure 1.2. In the next section, we discuss the commonly used behavioural transformations during embedded system synthesis.

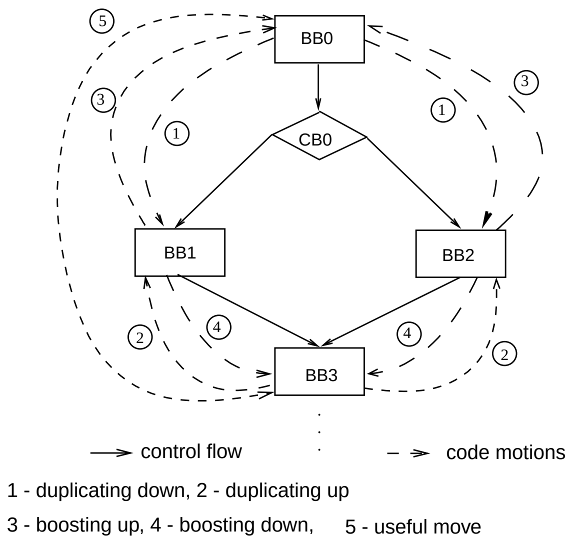

BB0
CB0
BB1
BB2
BB3
5
3
1
3
1
2
4
4
2
control flow
code motions
1 - duplicating down, 2 - duplicating up
3 - boosting up, 4 - boosting down,
5 - useful move

Figure 1.3: Various code motion techniques

1.2 Behavioural transformations

As discussed above, application of behavioural transformation techniques is a common practice during embedded system design. In course of devising the final implementation from the initial specification, a set of transformations may be carried out on the input behaviour targeting the optimal performance, energy and/or area on a given platform. In this section, we introduce several behavioural transformations that are commonly applied during embedded system design.

1.2.1 Code motion transformations

Code motion is a technique to improve the efficiency of a program by avoiding unnecessary re-computations (Knoop et al., 1992). The primary objective of code motion is the reduction of the number of computations at run-time. Secondary objective is the minimization of the lifetimes of temporary variables to avoid unnecessary register pressure. This can be achieved by moving operations beyond basic block boundaries. The code motion based transformation techniques can be classified into the following categories (Rim et al., 1995) using Fig 1.3: (*i*) Duplicating down refers to moving operations from a basic-block (BB) preceding a conditional block (CB) to both the BBs following the CB. *Reverse speculation* (Gupta et al., 2004b) and *lazy execution* (Rim et al., 1995) belong to this category. (*ii*) Duplicating up involves moving operations from a BB in which conditional branches merge to its preceding BBs in the conditional branches. *Conditional speculation* (Gupta et al., 2004b) and *branch balancing* (Gupta et al., 2003a) fall under this category. (*iii*) Boosting up moves operations from a BB within a conditional branch to the BB preceding the CB from which the conditional branch sprouts. Code motion techniques such as *speculation* (Gupta et al., 2004b) fall under this category. (*iv*) Boosting down moves operations from BBs within the conditional branches to a BB following the merging of the conditional branches (Rim et al., 1995). (*v*) Useful move refers to moving an operation to a control and data equivalent block. A code motion is said to be *non-uniform* when an operation moves from a BB preceding a CB to only one of the conditional branches, or vice-versa. Conversely, a code motion is said to be *uniform* when an operation moves to and fro both the conditional branches from a BB before or after the CB. Therefore, duplicating up and boosting down are inherently uniform code motions whereas *duplicating down and boosting up can be uniform as well as non-uniform*.

1.2.2 Loop transformations

As the name suggests, loop transformation techniques are used to increase instruction level parallelism, improve data locality and reduce overheads associated with executing loops of array-intensive applications (Bacon et al., 1994). Most execution time of a scientific program is spent on loops. Thus, a lot of compiler analysis and compiler optimization techniques have been developed to make the execution of loops faster. *Loop fission/distribution/splitting* attempts to break a loop into multiple loops over the same index range but each taking only a part of the loop body. The inverse transformation of loop fission is *fusion/jamming*. *Loop unrolling* replicates the body of a loop by some number of times (unrolling factor). Unrolling improves performance by reducing the number of times the loop condition is tested and by increasing instruction parallelism. *Loop skewing* takes a nested loop iterating over a multi-dimensional array, where each iteration of the inner loop depends on previous iterations, and rearranges its array accesses so that the only dependencies are between iterations of the outer loop. *Loop interchange/permutation* exchanges inner loops with outer loops. Such a transformation can improve locality of reference, depending on the array layout. *Loop tiling/blocking* reorganizes a loop to iterate over blocks of data sized to fit in the cache.

Loop unswitching moves a conditional from inside a loop to outside by duplicating the loop body. Some other important loop transformations are loop *reversal, spreading, peeling,* etc. (Bacon et al., 1994).

1.2.3 Arithmetic transformations

A compiler usually applies a set of techniques that transform the arithmetic expressions of the behaviours. Some of them are discussed here. *Common subexpression elimination:* In many cases, a set of computations will contain identical subexpressions. The compiler can compute the value of the subexpression once, store it, and reuse the stored result. *Constant propagation*: Typically programs contain many constants. By propagating them through the program, the compiler can do a significant amount of precomputation (Bacon et al., 1994). More importantly, the propagation reveals many opportunities for other optimization. *Constant folding* is a companion to constant propagation. When an expression contains an operation with constant values as operands, the compiler can replace the expression with the result. *Copy propagation*: Optimization such as common subexpression elimination may cause the same value to be copied several times. The compiler can propagate the original value through a variable and eliminate redundant copies. *Algebraic transformations:* The compiler can simplify arithmetic expressions by applying *algebraic rules such as associativity, commutativity and distributivity*. *Operator strength reduction:* The compiler can replace an expensive operator with an equivalent, less expensive operator. *Redundancy-eliminating transformations*: Compiler may remove unreachable and useless computations. A computation is unreachable if it is never executed. Removing it from the program will have no semantic effect on the instructions executed. A computation is useless if none of the outputs of the program are dependent on it.

1.2.4 High-level to RTL transformations

High-level synthesis (HLS) tools (Gajski et al., 1992) convert a high-level behavioural specification into an RTL level description. The HLS process consists of several interdependent subtasks such as compilation or pre-processing, scheduling, allocation and binding, and datapath and controller generation. In the first step of HLS, the behavioural description is compiled into an internal representation. This process usually includes a series of compiler like optimizations. Scheduling assigns operations of the behavioural description into control steps. Allocation chooses functional units and storage elements from the component library based on the design constraints. Binding assigns operations to functional units, variables to storage elements and data transfers to wires or buses such that data can be correctly moved around according to the scheduling. The final step of high-level synthesis is data-path and controller generation. Depending upon the scheduling and the binding information of the operations and the variables, proper interconnection between the data-path components is set up. Finally, a finite state machine (FSM) is generated to control all the micro-operations over the datapath. The RTL designs consist of a description of the datapath netlist and a controller FSM.

Power optimization can be performed on different levels of the design hierarchy. Lately, system level and high-level power optimization have received great attention (Ahuja et al., 2010; Jiong et al., 2004; Lakshminarayana et al., 1999; Musoll and Cortadella, 1995; Xing and Jong, 2007). Employing low power transformations at RTL designs, such as, restructuring multiplexer networks (to enhance data correlations and eliminate glitchy control signals), clocking control signals, and inserting selective rising/falling delays, clock gating, etc., has emerged as an important technique to minimize total power consumption of the circuits (Chandrakasan et al., 1995a, 1992; Raghunathan et al., 1999). In control-flow intensive designs, the multiplexer networks and registers dominate the total circuit power consumption. Also, the control logic can generate a significant amount of glitches at its outputs, which in turn propagate through the data path accounting for a large portion of the glitch power in the entire circuit. For data-flow intensive designs, the chaining of arithmetic functional units results in majority of dynamic power consumption.

1.2.5 Sequential to parallel transformations

Applications like multimedia, imaging, bioinformatics, and signal processing must achieve a high computational power with minimal energy consumption. Given these conflicting constraints, multiprocessor implementations not only deliver the necessary computational power, but also provide the required power efficiency. However, the performance gain achieved is dependent on how well the compiler can parallelize the given program and generate code for ‘matching’ the hardware parallelism. There are three types of parallelism of the sequential programs: (i) Loop-level parallelism: The iterations of a loop are distributed over multiple processors. (ii) Data-parallelism: data parallelism is achieved when each processor performs the same task on different pieces of distributed data. (iii) Task-level parallelism: Task parallelism focuses on distributing sub-tasks of a program across different parallel processors. The sub-tasks can be the subroutine calls, independent loops, or even independent basic blocks. This is the most used parallelization technique.

The parallel behaviour obtained from a sequential behaviour may undergo a parallel level code transformations. Parallel transformations manipulate the concurrency level of the parallel programs. The concurrency level of a program may not match the parallelism of the hardware, or vice-versa. In this case, the performance (in terms of total execution time, energy consumption, etc.) of that parallel program can be modified to suit the hardware by changing the concurrency level of the program. The transformations like *channel merging and splitting, process merging and splitting and computation migration, etc.*, are commonly used for this purpose.

1.3 Motivations and objectives

Verifying correctness of behavioural transformations, as described above, is an important step in ensuring dependability of embedded systems. Various analysis tools can prove the absence of certain kinds of errors at the source behaviour; however, if the compiler is not guaranteed to be correct, then no source-level guarantees can be safely transferred to the generated code. One of the most error prone parts of a compiler is its optimization phase. Many optimizations require an intricate sequence of complex transformations. Often these transformations interact in unexpected ways, leading to a combinatorial explosion in the number of cases that must be considered to ensure that the optimization phase is correct (Kundu et al., 2009). Vendors of mature ES design tools often talk of method which are “correct by construction". However, it should be noted that tools are always in a state of change as newer features have to be introduced to maintain competitiveness of the tools. Therefore, even with “mature" tools, there is a possibility of encountering a bug resulting in design flaws.

A degree of assurance is achieved by way of on extensive simulation and testing. The goal of simulation is to identify errors as early as possible in the design phase. In particular, for embedded systems, both the hardware and the software parts of the system must be simulated at the same time. Both simulation and testing suffer from being incomplete: each simulation run or each test evaluates the system performance for only a single set of operating conditions and input signals. For complex embedded systems, it is impossible to cover even a small fraction of the total operating space with simulations. Finally, testing is prohibitively expensive. Today building a test harness to simulate a component's environment is more expensive than building the component itself.

Formal verification can be used to provide guarantees of compiler correctness. It is an attractive alternative to traditional methods of testing and simulation, which for embedded systems, tend to be expensive, time consuming, and hopelessly inadequate, as argued above. There are two fundamental approaches of formal verification of compilers. The first approach proves that the steps of the compiler are *correct by construction*. In this setting, to prove that an optimization is correct, one must prove that for any input program the optimization produces a semantically equivalent program. The primary advantage of correct by construction techniques is that optimizations are known to be correct when the compiler is built, before they are run even once. Most of the techniques that provide correct by construction guarantees require user interaction (Kundu et al., 2009). Moreover, correct by construction proofs are harder to achieve because they must show that *any* application of the optimization is correct. Despite the fact that a significant amount of work has been carried out for verifying that synthesis steps are correct by construction, the state of the art techniques are still far from being able to prove automatically that the steps of the compilation process always produce correct designs (Kundu et al., 2010). However, even if one cannot prove a compiler to be correct by construction, one can at least show that, for each translation that a compiler performs, the output produced has the same behavior as the original behaviour. The second category of formal verification approach, called *translation validation*, consists of proving correctness each time an optimization step is invoked. Here, each time the compiler runs an optimization, an automated tool tries to prove that the original program and the corresponding optimized program are equivalent. Although this approach does not guarantee the correctness of the compilation process, it at least guarantees that any errors in translation will be caught when the particular steps of ES design tool are performed, preventing such errors from propagating any further in synthesis process. In this dissertation, we work on developing translation validation methodologies for several behavioural transformations applied during embedded systems.

1.3.1 Problem statements

The objective of this work is to show the correctness of several behavioural transformations that occur during embedded system design primarily using equivalence checking methods. Specifically, the following verification problems will be addressed:

Code motion transformations: Several code motion techniques such as speculation, reverse speculation, branch balancing, conditional speculation, etc., may be applied on the input behaviour which is in the form of a sequential code at the preprocessing stage of embedded system synthesis. The input behaviours are transformed significantly due to these transformations. One may also apply a combination of these transformation techniques. Moreover, arithmetic transformations may also be applied along with code motion transformations. It is, therefore, a non-trivial task to show the equivalence between the input behaviour and the transformed behaviour. The equivalence checking methods reported in the literature conventionally use path based approach whereupon for each path in a behaviour an equivalent path is identified in the transformed behaviour. Code motions can be of two types namely, uniform and non-uniform code motions. The former moves a code segment over both branches following a branching block; in contrast, the latter category of code motions move code segments over only one branch of the BB. The methods reported in the literature can verify only uniform code motions. A common verification mechanism for both kinds of code motion transformations is worth exploring.

High-level to RTL transformations: During high-level synthesis, the input high-level behaviour is transformed to an output RTL consisting of a datapath, which is merely a structural description, and a controller, represented as a finite state machine (FSM). The controller invokes a control assertion pattern (CAP) in each control step to execute all the required data-transfers and proper operations in the FUs. The results of the relational operations (i.e. the status signals) are the inputs to the controller. The state transitions in the controller FSM depend on these status signals. The input behaviour is in a higher abstraction level compared to that of the output RTL behaviour. The verification method, therefore, should first analyze the CAP vis-a-vis the datapath structure to identify the RT-operations from the output RTL behaviour to compare it with the input high-level behaviour. The data transformations of the high-level behaviour, however, may be implemented in pipelined or multicycle functional units which may execute over more than one FSM control state. Therefore, a state-wise analysis of CAP vis-a-vis datapath inter-connections may not suffice to verify multicycle and pipelined operations. The verification task of this phase, therefore, should accommodate all the intricacies of the RTL designs to show the equivalence between the high-level behaviour and the RTL behaviour.

RTL transformations: The low power RTL transformations primarily bring about modifications at a much lower level of abstraction involving intricate details regarding restructuring of the datapaths, control logic and routing of the control signals. Accordingly, in a typical industry scenario, an RTL or architectural low power transformation implies a full cost of simulation based validation, which can extend to many months (Viswanath et al., 2009). Formal verification methods, not necessarily compromising the details through abstraction, is a desirable goal. One of the objectives of this work is showing equivalence of two RTL designs, one obtained from the other by applying low power RTL transformations.

Loop transformations: Signal processing and multimedia applications are data dominated in nature. Several loop based transformation techniques such as un-switching, reordering, skewing, tiling, unrolling, etc., may be applied at the preprocessing stage. These transformations are applied in order to reorder and restructure the loop body to improve the spatial and temporal locality of the accessed data. The loop transformation techniques can be of two types, structure preserving and structure modifying. The former admits a clear mapping of control and data values in the transformed behaviour to the corresponding control and data values in the source behaviour; the latter does not admit such a mapping (Zuck et al., 2005). In addition, arithmetic transformations may also be applied along with loop transformations. The verification method should be strong enough to handle as many loop transformations and arithmetic transformations as possible applied on data dominated behaviours.

Sequential to parallel behaviour transformation: The functional specification, which relies on a single-threaded sequential code, is not easy to deploy on highly concurrent heterogeneous multi-processor systems. A parallel model of computation (MoC) may be used as the programming model. However, writing an application depicting concurrency is time consuming and error prone which conflicts with the low time-to-market requirement of the present day embedded systems. So, an automated tool that converts a sequential code to its equivalent concurrent behaviour is employed. In the case of streaming applications, Kahn process network (KPN) (Kahn, 1974) model of computation is often used. The KPN is a deterministic parallel MoC that explicitly specifies tasks as processes and distributed memory units as FIFO channels. The verification task involves showing the equivalence between a sequential behaviour and its corresponding parallel KPN behaviour. Obviously, the overall data transformation of the KPN behaviour can only be captured if the global data dependencies are captured. The challenge lies in capturing the data dependencies spread over all the concurrent KPN processes independent of their possible interleaving.

Parallel level transformations: The parallel process network model that are obtained from the sequential behaviour in an automated way or manually may not be suitable for the target architectural platform. Moreover, the overall achieved performance obtained as a result of mapping may not be satisfactory from the point of view of the data throughput and/or memory usage. In this case, it is necessary to manipulate the amount of concurrency in the functional model. Too little concurrency may not be desirable since it may not completely make use of the architectural capabilities. Too much concurrency, in contrast, may require too large an overhead to eventually manage in the architecture, which may increase the overall latency. The transformation techniques like process splitting, channel merging, process clustering, and unfolding and skewing may be used to control the concurrency in the KPN behaviour according to the architectural constraints. The verification task of this phase, therefore, is to show the equivalence between two KPN behaviours. The challenge remains the same as in equivalence checking of sequential to parallel transformation namely, capturing the global dependencies irrespective of the interleaving of both the input and the output concurrent behaviours. Hence, it is worth examining whether a method capable of validating sequential to parallel transformations should suffice for the commonly used parallel level transformations.

1.4 Contributions

Verification of code motion transformations: A formal verification method for checking correctness of code motion techniques is presented. Finite state machine models with datapath (FSMDs) have been used to represent the input and the output behaviours. The method introduces cutpoints in one FSMD, visualizes its computations as concatenation of paths from cutpoints to cutpoints, and then identifies equivalent finite path segments in the other FSMD; the process is then repeated with the FSMDs interchanged. A path is extended when its equivalent path cannot be found in the other FSMD. However, a path cannot be extended beyond loop boundaries. Our method is capable of verifying both uniform and non-uniform code motion techniques. It has been underlined in this work that for non-uniform code motions, identifying equivalent path segments involves model checking of some data-flow properties. Our method automatically identifies the situations where such properties are needed to be checked during equivalence checking, generates the appropriate properties, and invokes the model checking tool NuSMV to verify them. The correctness and the complexity of the method have been dealt with. Experimental results demonstrate the effectiveness of the method.

Verification of RTL generation and RTL transformations: A formal verification method of the RTL generation phase of a high-level synthesis (HLS) process is presented in (Karfa, 2007). The goal is achieved in two steps. In the first step, the datapath interconnection and the controller FSM description generated by a high-level synthesis process are analyzed to obtain the register transfer (RT) operations executed in the datapath for a given control assertion pattern in each control step. In the second step, an equivalence checking method is deployed to establish equivalence between the input and the output behaviours of this phase. A rewriting method has been developed for the first step. Our method is strong enough to handle pipelined and multicyle operations, if any, spanning over several states. The correctness (termination, soundness and completeness) and complexity of the presented method have been treated formally. The experimental results on several HLS benchmarks are presented.

In order to verify RTL transformations, we model both the RTLs as FSMDs and then apply our FSMD based equivalence checking method to show the equivalence. In this work, we analyze several commonly used RTL low power transformation techniques and show that our verification method can handle those transformations.

Verification of loop and arithmetic transformations of array intensive behaviours: We propose a formal verification method for checking correctness of loop transformations and arithmetic transformations applied on the array and loop intensive behaviours in design of area/energy efficient systems in the domain of multimedia and signal processing applications. Loop transformations hinge more on data dependence and index space of the arrays than on control flow of the behaviour. Hence, array data dependence graphs (ADDGs), proposed by Shashidhar (Shashidhar, 2008), are used for representing array intensive behaviours. Shashidhar (Shashidhar, 2008) proposed an ADDG based equivalence checking method to validate the loop transformations. Possible enhancements of his method to handle associative and commutative transformations are also discussed in (Shashidhar, 2008; Shashidhar et al., 2005a). We redefine the equivalence of ADDGs in this work to verify loop transformations along with a wide range of arithmetic transformations. We also propose a normalization method for array intensive integer arithmetic expressions. The equivalence checking method relies on this normalization technique and some simplification rules to handle arithmetic transformations over arrays. Correctness and complexity of the method have been dealt with. Experimental results on several test cases are presented.

Verification of Parallelizing transformations: A formal verification method for checking correctness of parallelizing transformations of sequential behaviours is presented. The parallel behaviours are represented as Kahn process networks (KPNs). We describe a mechanism to represent a KPN behaviour comprising inherently parallel processes as an ADDG. We also present a proof of correctness of the construction mechanism. We then apply our ADDG based equivalence checking method to establish the equivalence between the initial sequential behaviour and the KPN behaviour. The key aspect of our scheme is to model a KPN behaviour as an ADDG. We then show how the ADDG based modelling helps us detect deadlocks in KPN behaviours. Experimental results supporting this scheme are provided.

We next address the problem of verifying transformations of parallel behaviours. We assume that the parallel behaviours and their transformed versions are both available as KPN models. The KPN to ADDG transformation scheme is applied for this purpose. Specifically, we model both the input and the transformed KPNs as ADDGs and apply our ADDG based equivalence checking method for establishing the equivalence between the two KPNs. We then show that the commonly used parallel transformations techniques can be verified in our verification framework. Experimental results are presented.

1.5 Organization of the thesis

The rest of the thesis is organized in the following manner.

Chapter 2 provides a detailed literature survey on the applications of commonly used behavioural transformations and their existing verification methods. In the process, it identifies the limitations of the state of the art verification methods and underlines the objectives of the thesis.

Chapter 3 discusses a path extension based equivalence checking of code motion transformations. The issues addressed in the chapter is illustrated first. The FSMD model is then introduced. A normalization procedure over integer arithmetic is given next. The notion of path extension and its enhancement for verifying non-uniform code motions are described next. The verification method is then given. The chapter also provides a detailed theoretical analysis of the method and some experimental results with the implementation.

Chapter 4 discusses the high-level to RTL translation verification method. The basic construction mechanism of the FSMDs from the RTL designs is discussed first. How the basic method is enhanced to handle multicyle, pipelined and chained operations is given next. The overall construction mechanism is then discussed. Termination, soundness and completeness of the method have been proved and the complexity of the method analyzed. The verification of the data-path and the controller during FSMD construction is given next. The chapter then discusses the applicability of this method to verify RTL level transformations and finally, provides some experimental results on this method.

Chapter 5 discusses the verification of loop and arithmetic transformations of array intensive behaviours. The ADDG model and its different entities are formally defined and elaborated with examples first. The method to obtain an ADDG from a sequential behaviour is then presented. A normalization method and some simplification rules of normalized expressions for array intensive arithmetic expressions are introduced. The chapter defines the notion of slice based equivalence of ADDGs next. The correctness and complexity of the method have been analyzed subsequently. The chapter finally provides some experimental results and error diagnoses of the presented method.

Chapter 6 discusses the verification of sequential to parallel and parallel to parallel code transformations. The objectives of the work are illustrated first. The verification framework is given next. The chapter then provides a mechanism of modelling a KPN as an ADDG. The proof of correctness of this translation is given next. The chapter then discusses the commonly applied KPN level transformations and shows that our verification framework is strong enough to verify those transformations. Finally, some experimental results are provided.

Chapter 7 concludes our study in the domain of verification of behaviour transformations during embedded system design and discusses potential future research directions in this field.

Chapter 2

Literature Survey

An overview of important research contributions in the area of behavioural transformations during embedded system design is provided in this chapter. For each transformation, (i) We first study several applications of behavioural transformations during embedded system design. The objective of this study is to establish the relevance of these transformations in embedded system design. (ii) we then survey existing verification methodologies for each of behavioural transformations. The objective of this study is to find the limitations of the existing verification methodologies and to identify the verification problems which have been addressed in this thesis.

2.1 Code motion transformations

2.1.1 Applications of code motion transformations

Application of code motion techniques is well studied in literature in the context of parallelizing compilers (Aho et al., 1987; Fisher, 1981; Gupta and Soffa, 1990; Hwu et al., 1993; Knoop et al., 1992; Lam and Wilson, 1992; Muchnick, 1997; Nicolau and Novack, 1993; Ruthing et al., 2000). Efforts have recently been made to find its effect during synthesis, specially during scheduling in high-level synthesis. In the following, we study the applications of code motion techniques in the context of high-level synthesis.

Santos et al. (Dos Santos et al., 2000; Dos. Santos and Jress, 1999) and Rim et al. (Rim et al., 1995) support generalized code motions during scheduling in synthesis systems, whereby operations can be moved globally irrespective of their position in the input. The costs of a set of solutions for the given design are evaluated first and then the solution with the lowest cost is selected. Santos et al. proposed a pruning technique to select the lowest cost solution from a set of solutions. Their technique explore a smaller number of solutions which effectively reduce the search time.

Speculative execution refers to the execution of parts of a computation before the execution of the conditional operations that decide whether they need to be executed. The paper (Lakshminarayana et al., 2000) presents techniques to integrate speculative execution into scheduling during high-level synthesis. This work shows that the paths for speculation need to be decided dynamically according to the criticality of individual operation and the availability of resources in order to obtain maximum benefits. Their method has been integrated into the Wavesched tool (Lakshminarayana et al., 1997).

A global scheduling technique for superscalar and VLIW processors was presented in (Moon and Ebcioğlu, 1992). This technique parallelizes sequential code by eliminating anti-dependence (i.e., write after read) and output dependence (i.e., write after write) by renaming registers. The code motions are applied globally by keeping a data flow attribute at the beginning of each basic block which indicates what operations are available for moving up through this basic block.

In (Johnson, 2004), the register allocation phase and the code motion methods are combined to obtain a better scheduling of instructions with less number of registers. Register allocation can artificially constrain instruction scheduling, while the instruction scheduler can produce a schedule that forces a poor register allocation. The method proposed in this work tried to overcome this limitation by combining these two phases of high-level synthesis.

In (Cordone et al., 2006), control and data dependence graph (CDFG) is used as an intermediate representation which provides the possibility of extracting the maximum parallelism from the behaviour. This work combines the speculative code motion techniques and parallelizing techniques to improve scheduling of control flow intensive behaviours.

The effectiveness of traditional compiler techniques employed in high-level synthesis of synchronous circuits is studied for asynchronous circuit synthesis in (Zaman-Zadeh et al., 2009). It has been shown that the transformations like speculation, loop invariant code motion and condition expansion are applicable in decreasing mass of handshaking circuits and intermediate modules.

Gupta et al. (Gupta et al., 2003b, 2004b,c) pioneered the work of applying code motions to improve results of high-Level synthesis. They have used a set of speculative code motion transformations that enable movement of operations through, beyond, and into conditionals with the objective of maximizing performance. *Speculation, reverse speculation, early condition execution, conditional speculation techniques* are introduced by them in (Gupta et al., 2004b, 2001a,b). They present a scheduling heuristic that guides these code motions and improves scheduling results (in terms of schedule length and FSM states) and logic synthesis results (in terms of circuit area and delay) by up to 50 percent. In (Gupta et al., 2003a,b), two novel strategies are presented to increase the scope for application of speculative code motions: (a) Adding scheduling steps dynamically during scheduling to conditional branches with fewer scheduling steps. This increases the opportunities to apply code motions such as conditional speculation that duplicate operations into the branches of a conditional block. (b) Determining if an operation can be conditionally speculated into multiple basic blocks either by using existing idle resources or by creating new scheduling steps. These strategies lead to balancing of the number of steps in the conditional branches without increasing the longest path through the conditional block. In (Gupta et al., 2002), a new approach called *dynamic common sub-expression elimination (CSE)* is introduced. It dynamically eliminates common sub-expressions based on new opportunities created during scheduling of control-intensive designs. Classical CSE techniques fail to eliminate several common sub-expressions in control-intensive designs due to the presence of a complex mix of control and data-flow. Aggressive speculative code motions employed to schedule control intensive designs often re-order, speculate and duplicate operations, changing thereby the control flow between the operations with common sub-expressions. This leads to new opportunities for applying CSE dynamically. They also present a technique called *loop shifting* in (Gupta et al., 2004a) that incrementally exploits loop level parallelism across iterations by shifting and compacting operations across loop iterations. It has been shown through experimentation that loop shifting is particularly effective for the synthesis of designs with complex control especially when resource utilization is already high and/or under tight resource constraints. Their code motion techniques and heuristics have been implemented in the SPARK high-level synthesis framework (Gupta et al., 2003c).

2.1.2 Verification of code motion transformations

A formal verification of scheduling process using finite state machines with data-path (FSMD) is reported in (Kim et al., 2004). In this paper, break-points are introduced in both the FSMDs followed by construction of the respective path sets. Each path of one set is then shown to be equivalent to some path of the other set. This approach necessities that the path structure of the input FSMD is not disturbed by the scheduling algorithm and code has not moved beyond basic block boundaries. in the sense that the respective path sets obtained from the break points are assumed to be bijective. This property, however, does not necessarily hold because the scheduler may merge the segments of the original specification into one segment or distribute operations of a segment over various segments for optimization of time steps.

An automatic verification of scheduling by using symbolic simulation of labeled segments of behavioural descriptions has been proposed in (Eveking et al., 1999). In this paper, both the inputs to the verifier, namely the specification and the implementation, are represented in the *Language of Labeled Segments (LLS)*. Two labeled segments S_1 and S_2 are bisimilar iff the same data-operations are performed in them and control is transformed to the bisimilar segments. The method described in this paper transforms the original description into one which is bisimilar with the scheduled description.

A Petri net based verification method for checking the correctness of algorithmic transformations and scheduling process in HLS is proposed in (Chiang and Dung, 2007). The initial behaviour is converted first into a Petri net model which is expressed by a Petri net characteristic matrix. Based on the input behaviours, they extract the initial firing pattern. If there exists at least one candidate who can allow the firing sequence to execute legally, then the HLS result is claimed as a correct solution.

All these verification approaches, however, are well suited for basic-block based scheduling (Jain et al., 1991; Lee et al., 1989a) where the operations are not moved across the basic-block boundaries or the path-structure of the input behaviour does not modify due to scheduling. These techniques are not applicable to the verification of code motion techniques since code may be moved from one basic block to other basic blocks due to code motions.

Some recent works, such as, (Karfa, 2007; Kim and Mansouri, 2008; Kundu et al., 2008, 2010) target verification of code motion techniques. Specifically, a path recomposition based FSMD equivalence checking has been reported in (Kim and Mansouri, 2008) to verify speculative code motions. The correctness conditions are formulated in higher-order logic and verified using the PVS theorem prover. Their path recomposition over conditional blocks fails if non-uniform code motion transformations are applied by the scheduler. In (Kundu et al., 2008, 2010), a translation validation approach is proposed for high-level synthesis. Bisimulation relation approach is used to prove equivalence in this work. Their method automatically establishes a bisimulation relation that states which points in the initial behaviour are related to which points in the scheduled behaviour. This method apparently fails to find the bisimulation relation if codes before a conditional block are not moved to all branches of the conditional block. This method also fails when the control structure of the initial program is transformed by the path-based scheduler (Camposano, 1991). We proposed an equivalence checking method for scheduling verification in (Karfa, 2007). This method is applicable even when the control structure of the input behaviour has been modified by the scheduler. It has been shown that this method can verify uniform code motion techniques. This method provides false-negative result for non-uniform code motions.

The above discussion suggests that some of the existing methods can verify uniform code motions. None of existing methods, however, works for non-uniform code motion. It would be desirable to have an equivalence checking method for verifying both uniform and non-uniform code motion techniques.

2.2 High-level to RTL and RTL transformations

2.2.1 Applications of High-level to RTL and RTL transformations

Staring from early tools like MAHA (Parker et al., 1986), HAL (Paulin and Knight, 1987), STAR (Tsai and Hsu, 1992) to recent days' tools such as SPARK (Gupta et al., 2003c), SAST (Karfa et al., 2005), Catapult (Graphics, 2006), Synphony (Synopsys, 2011), a numerous tools have been developed with different optimization goals. Many of them target minimization of time steps in the scheduling phase of high-level synthesis. Many others target register optimization and some others target datapath and controller optimizations. Some of the notable methods are discussed here.

Early works focused on scheduling steps of high-level synthesis. The simplest one schedules all the operations as soon as possible (ASAP) (Trickey, 1987; Tseng and Siewiorek, 1986). The opposite approach is to schedule the operations as late as possible (ALAP). Several other scheduling approaches are force-directed scheduling (Paulin and Knight, 1987), integer linear program based formulation of scheduling (Lee et al., 1989b), list scheduling (Sllame and Drabek, 2002), genetic algorithm based scheduling (Mandal and Chakrabarti, 2003). We have already discussed applications of several code motion techniques during scheduling in the previous subsection.

Register optimizations are primarily performed by clique partition technique. Tseng et al. (Tseng and Siewiorek, 1986) use clique partitioning heuristics to find a clique cover for a module allocation graph. Paulin et al. (Paulin and Knight, 1989) perform exhaustive weight-directed clique partitioning of a register compatibility graph to find the solution with the lowest combined register and interconnect costs.

Many approaches target datapath and controller optimizations. The work of Kim (Kim and Liu, 2010), for example, targets reduction of the multiplexer inputs and shortening of the total length of global interconnects. Specifically, this method maximizes the interconnect sharing among FUs and registers. The datapath interconnection optimization problem is mapped to a minimal cost maximal flow algorithm in (Zhu and Jong, 2002). The algorithm proposed in (Kastner et al., 2006) uses floorplan information to optimize data communication of the variables. SAST (Karfa et al., 2005; Mandal and Zimmer, 2000) produces structured architectures which avoid random inter-connections in the datapaths. In (Lin and Jha, 2005), the datapath interconnections are optimized for power during high-level synthesis. This method not only reduces datapath unit power consumption in the resultant RTL but also optimizes interconnects for power.

A set of glitch power reduction techniques based on minimizing propagation of glitches in the RTL circuit is proposed in (Raghunathan et al., 1999). These techniques include restructuring multiplexer networks (to enhance data correlations and eliminate glitchy control signals), clocking control signals, and inserting selective rising/falling delays, in order to kill the propagation of glitches from control as well as data signals. More than twenty architectural and computational transformation techniques are presented in (Chandrakasan et al., 1995b) for minimizing power consumption in application specific datapath intensive circuits. Several other RTL power management schemes are presented in (Ahuja et al., 2010; Jiong et al., 2004; Lakshminarayana et al., 1999; Xing and Jong, 2007).

2.2.2 Verification of High-level to RTL and RTL transformations

A set of high-level synthesis systems is validated using formally verified transformations in (Radhakrishnan et al., 2000). This tool examines the output of a high-level synthesis system and derives a sequence of behaviour-preserving (correct) transformations (witness) that leads to the same effect as the applied HLS algorithm. If every transformation, identical in the derived sequence, is applied in the presence of a set of preconditions (which are proved to lead to a correct design), then the resulting RTL design is correct. In (Rajan, 1995), both the specification at the behavioural level and the implementation at the RTL level are encoded in SIL (Krol et al., 1992). A small set of properties (axioms) corresponding to the SIL graph is asserted to be true. These axioms capture the general notion of refinement of the CDFG used in various synthesis frameworks. In order to compare the behavioral and the RTL descriptions in a uniform way, a framework for verification and reasoning of the descriptions based on mapping to virtual datapaths/controllers from these descriptions is developed in (Fujita, 2005). Once those mappings have been established, the real comparison can be based on the data transfer analysis controlled by finite state machines. In (Mansouri and Vemuri, 1998), the technique of determining the correctness of the RTL design depends upon comparing the values of certain critical specification variables in certain critical behavioural states with those of certain critical RTL registers in certain critical controller states, provided they match at the start state. This approach, however, necessitates that the control flow branches in the behaviour specification are preserved and no new control flow branches are introduced.

As discussed above, some end-to-end validation approaches of high-level synthesis are proposed in the literature. The end-to-end HLS verification techniques are not efficient enough as it is not only error prone but also unable to find the exact subtask in which the error occurs. Also, these techniques are not sophisticated enough to handle the advanced transformations applied in each phase of HLS. For these reason, verification of different phases of HLS are proposed in literature. Verification techniques for the scheduling phase are discussed in the previous subsection. Since, high-level behaviours are actually mapped to RTL design in datapath and controller (i.e., RTL) generation phase, we review the verification techniques of this phase of high-level synthesis next.

The allocation and binding phase and the datapath and controller generation phase have been verified together in (Borrione et al., 2000) using the FSMD model. Demonstrating the equivalence of the FSMD obtained by functional composition of two parts of the implementation, the control part and the operative part, with the scheduled FSMD accomplishes the functional verification. In (Ashar et al., 1998), the synthesis of datapath interconnect and control is verified as follows: The operations in each state of the scheduled behaviour are converted to an equivalent *structured graph (SG)* having the hardware components as vertices and connectivity among the components as edges. It then shows the equivalence between the SSG (scheduler SG) and the RSG (RTL SG) by symbolic simulation to ensure that the datapath supports the RT-operations of that state. However, the state-wise interaction of the controller and the datapath through the status and control lines has not been verified; also, multicyle and pipelined operations have not been considered.

The low power transformations represent an important aspect of power related optimizations. However, the main obstacle of applying low power transformations at RTL designs is the difficulties of its verification problem. In a typical industry scenario, an RTL or architectural low power transformation implies a full cost of dynamic validation, which can extend to many months (Viswanath et al., 2009). Therefore, an automated formal verification method for low power transformations at RTL designs has tremendous practical importance. A dedicated rewriting method for verification of low power transformations in RTL has been proposed in (Viswanath et al., 2009) where low power transformations are verified based on transformation specific rules. The completeness of this method, therefore, depends on the availability of the transformation rules. Also, a combination of low power transformations may be applied on the RTL designs. In such cases, transformation rules for individual transformation may not be able to establish the equivalence between the input RTL and the transformed RTL.

One of the objectives of this work is to develop an equivalence checking method between a high-level behaviour and its corresponding RTL behaviour which can handle pipelined and multicyle operations. Moreover, the method can be easily tuned to verify the RTL level transformations.

2.3 Loop transformations and arithmetic transformations

2.3.1 Applications of loop transformations

Loop transformations along with algebraic and arithmetic transformations are applied extensively in the domain of multimedia and signal processing applications. These transformations can be automatic, semi-automatic or manual. In the following, we study several applications of loop transformations techniques during embedded system design.

Kandemir et al. studied the effects of loop transformations on system power in their several works. In (Kandemir et al., 2001), they studied the impact of loop tiling, loop unrolling, loop fusion, loop fission and scalar expansion on energy consumption. In (Kandemir et al., 2005), they demonstrate that conventional data locality oriented code transformations are not sufficient for minimizing disk power consumption. Instead, they propose a disk layout aware application optimization strategy that uses both code restructuring and data locality optimization. They focus on three optimizations namely, loop fusion/fission, loop tiling, and linear optimizations for code restructuring and also discuss how these transformations can be combined under a unified optimizer that targets disk power management. They show how code and data optimizations help to reduce memory energy consumption for embedded applications with regular data access patterns on an MPSoC architecture with a banked memory system in (Kandemir, 2006). This is achieved by ensuring bank locality, which means that each processor localizes its accesses into a small set of banks in a given time period. They also propose a novel memory-conscious loop parallelization strategy with the objective of minimizing the data memory requirements of processors in (Xue et al., 2007). The work in (Kadayif and Kandemir, 2005) presents a data space-oriented tiling approach (DST). In this strategy, the data space is logically divided into chunks (called data tiles) and each data tile is processed in turn. Since a data space is common across all nests that access it, DST can potentially achieve better results than traditional iteration space (loop) tiling by exploiting inter-nest data locality. Improving data locality not only improves effective memory access time but also reduces memory system energy consumption due to data references. The paper (Li and Kandemir, 2005) takes a more global approach to identify data locality problem and proposes a compiler driven data locality optimization strategy in the context of embedded MPSOCs. An important characteristic of this approach is that, in deciding the workloads of the processors (i.e., in parallelizing the application) it considers all the loop nests in the application simultaneously. Focusing on an embedded chip multiprocessor and array-intensive applications, the work in (Chen et al., 2006a) demonstrates how reliability against transient errors can be improved without impacting execution time by utilizing idle processors for duplicating some of the computations of the active processors. It also shows how the balance between power saving and reliability improvement can be achieved using a metric called the energy-delay-fallibility product.

Bouchebaba et al. identified that general loop-based techniques fail to capture the interactions between the different loop nests in the application. In (Bouchebaba et al., 2007), they propose a technique to reduce cache misses and cache size for multimedia applications running on MPSoCs. Loop fusion and tiling are used to reduce cache misses, while a buffer allocation strategy is exploited to reduce the required cache size. The loop tiling exploration is further extended in (Zhang and Kurdahi, 2007) to also accommodate dependence-free arrays. They propose an input-conscious tiling scheme for off-chip memory access optimization. They show that the input arrays are as important as the arrays with data dependencies when the focus is on memory access optimization instead of parallelism extraction.

A method to minimize the total energy while satisfying the performance requirements for application with multi-dimensional nested loops was proposed in (Karakoy, 2005). They have shown that an adaptive loop parallelization strategy combined with idle processor shut down and pre-activation can be very effective in reducing energy consumption without increasing execution time. The objective of the paper (Qiu et al., 2008) is also the same as that of (Karakoy, 2005). However, they apply loop fusion and multi-functional unit scheduling techniques to achieve that.

In (Ghodrat et al., 2009), a novel loop transformation technique optimizes loops containing nested conditional blocks. Specifically, the transformation takes advantage of the fact that the Boolean value of the conditional expression, determining the true/false paths, can be statically analyzed using a novel interval analysis technique that can evaluate conditional expressions in the general polynomial form. Results from interval analysis combined with loop dependency information is used to partition the iteration space of the nested loop. This technique is particularly well suited for optimizing embedded compilers, where an increase in compilation time is acceptable in exchange for significant performance increase.

A survey on application of loop transformations in data and memory optimization in embedded system can be found in (Panda et al., 2001). The IMEC group (Catthoor et al., 1998; Palkovic et al., 2009) pioneered the work on program transformations to reduce energy consumption in data dominated embedded applications. In (Fraboulet et al., 2001), loop fusion technique is used to optimize multimedia applications before the hardware/software partitioning to reduce the use of temporary arrays. Several other loop transformation techniques and their effects on embedded system design may be found in (Brandolese et al., 2004; Ghodrat et al., 2008; Šimunić et al., 2000; Zhu et al., 2004).

2.3.2 Applications of arithmetic transformations

Several arithmetic transformations based on algebraic properties of the operator such as associativity, commutativity and distributivity, arithmetic expression simplification, constant folding, common sub-expression elimination, renaming, dead code elimination, copy propagation and operator strength reduction, etc. are applied regularly during embedded system design. Application of retiming, algebraic and redundancy manipulation transformations to improve the performance of embedded systems is proposed in (Potkonjak et al., 1993). They introduced a new negative retiming technique to enable algebraic transformations to improve latency/throughput. In (Zory and Coelho, 1998), the use of algebraic transformations to improve the performance of computationally intensive applications are suggested. In this paper, they investigate source-to-source algebraic transformations to minimize the execution time of expression evaluation on modern computer architectures by choosing a better way to compute the expressions. Operation cost minimization by loop-invariant code motion and operator strength reduction is proposed in (Gupta et al., 2000) to achieve minimal code execution within loops and reduced operator strengths. Application of algebraic transformations to minimize criticial path length in the domain of computationally intensive applications is proposed in (Landwehr and Marwedel, 1997). Apart from standard algebraic transformations such as commutativity, associativity and distributivity, they also introduce two hardware related transformations based on operator strength reduction and constant unfolding. The work in (Gupta et al., 2000) explores applicability and effectiveness of source-level optimizations for address computations in the context of multimedia applications. The authors propose two processor-independent source-level optimization techniques, namely, global scope operation cost minimization complemented with loop-invariant code hoisting, and non-linear operator strength reduction. The transformations attempt to achieve minimal code execution within loops and reduced operator strengths. A set of transformations such as common subexpression elimination, renaming, dead code elimination and copy propagation are applied along with code motion transformations in the pre-synthesis and scheduling phase of high-level synthesis in the SPARK tool (Gupta et al., 2003c, 2004c). The potential of arithmetic transformations on FPGAs is studied in (E. Özer and Gregg, 2003). It has been shown that operator strength reduction and storage reuse reduce the area of the circuit and hence the power consumption in FPGA. The transformations like height reduction and variable renaming reduce the total number of clock cycles required to execute the programs in FPGA whereas expression splitting and resource sharing reduce the clock period of the circuits.

2.3.3 Verification of loop and arithmetic transformations

A number of research works have been carried out on verification of loop transformations on array intensive programs. Some of these target transformation specific verification rules. Pnueli et al. proposed permutation rules for verification of loop interchange, skewing, tiling, reversal transformations in their translation validation approach (Zuck et al., 2003), (Zuck et al., 2005). The rule set is further enhanced in (Hu et al., 2005), (Barrett et al., 2005). The main drawback of this approach is that the method had to rely on the hint provided by the synthesis tool. The verifier needs the transformations that have been applied and the order in which they have been applied from the synthesis tool. Also, completeness of the verifier depends on the completeness of the rule set and therefore enhancement of the repository of transformations necessiates enhancement of the rule set.

A method called fractal symbolic analysis has been proposed in (Menon et al., 2003). The idea is to reduce the gap between the source and the transformed behaviour by repeatedly applying simplification rules until the two behaviours become similar enough to allow a proof by symbolic analysis. The rules are similar to the ones proposed by Pnueli et al. The power of this method again depends on the availability of the rule set.

Samsom et al. has developed a fully automatic verification method for loop transformations. They have used data dependence analysis and show the preservation of the dependencies in the original and in the transformed program (Samsom et al., 1995). The program representation used in this work allows only a statement-level equivalence checking between the programs. Therefore, this method cannot handle arithmetic transformation. It is common that data-flow transformations, such as expression propagations and algebraic transformations, are applied in conjunction with or prior to applying loop transformations in order to reduce the constraints for loop transformations. Therefore, direct correspondence between the statement classes of the original and the transformed programs does not always hold as required by Samsom's method.

Shashidhar et al. (Shashidhar, 2008; Shashidhar et al., 2002, 2005a,b) consider a restricted class of programs which must have static control-flow, affine indices and bounds, uniform recurrence and single assignment form. They have proposed an equivalence checking method for verification of loop and data-flow transformations.

```
A[0] = in[0];
for(i=1; i<100; i++)
   A[i] = f(A[i/2]);
```

Figure 2.1: A program with non-uniform recurrence on array A

```
for(i=0; i<100; i++)
   if(i < in1[i])
      out[i] = f(in2[i + in3[i + 1]]);
```

Figure 2.2: A program with data dependent control and array access

An array data dependence graph (ADDG) model was used in their work to model the loop-based array intensive programs. These works are promising in the sense that they are capable of handling most of the loop transformation techniques without taking any information from the synthesis tools.

The main limitations of the ADDG based method are its inability to handle the following cases (i) non-uniform recurrence (figure 2.1), (ii) data-dependent assignments and accesses (figure 2.2) and (iii) arithmetic transformations (2.3). In figure 2.1, the non-uniform recurrence involves the access sequence $i, i/2, 1/2^2$ and so on (in contrast to uniform recurrence access sequence such as $i, i-2, i-4$, etc). In figure 2.2, execution of the if statement depends on the array *in*1 and access to the elements of the array *in*2 depends of array *in*3. In figure 2.3, the transformed program is obtained from the original one by application of commutative and distributive transformations.

```
for(i=1; i<100; i++)
   A[i] = B[i + 1] x C[i] + C[i] x D[i];
              (a)


for(i=1; i<100; i++)
   A[i] = C[i] x (B[i + 1]  + D[i]);
              (b)
```

Figure 2.3: Commutative and distributive transformations: (a) original program; (b) transformed program

The method proposed in (Verdoolaege et al., 2009) extends the ADDG model to a dependence graph to handle non-uniform recurrences. This method is further extended in (Verdoolaege et al., 2010) to verify data-dependent assignments and accesses also. Both the methods proposed in (Verdoolaege et al., 2009) and (Verdoolaege et al., 2010) can verify the associative and commutative transformations. It has also been discussed in (Shashidhar, 2008) how the basic ADDG based method can be extended to handle associative and commutative transformations.

All the above methods, however, fail if the transformed behaviour is obtained from the original behaviour by application of arithmetic transformations such as, distributive transformations, arithmetic expression simplification, constant unfolding, common sub-expression elimination, etc., along with loop transformations. The definition of equivalence of ADDGs proposed by Shashidhar et al. cannot be extended easily (as in the cases of commutative and associative transformations) to handle these arithmetic transformations. In this dissertation, we consider the same class of programs considered by Shashidhar et al. and their ADDG based modelling of programs. It would be desirable to develop an equivalence checking method of array intensive programs based on Shashidhar's work which can handle all kinds of loop transformations along with most of the arithmetic transformations.

2.4 Parallelizing transformations

2.4.1 Applications of parallelizing transformations

Sequential to parallel code transformation techniques, in general, comprise two tasks – partitioning the sequential code into concurrent tasks and inserting appropriate communication channels among them. Many of the reported techniques (Freisleben and Kielmann, 1995; Hall et al., 1996; Wolf and Lam, 1991) exploit the data parallelism among different iterations of a loop to split the sequential behaviour into concurrent tasks and execute them in parallel on different processors. In (Freisleben and Kielmann, 1995), for example, a system is presented that automatically transforms sequential divide and conquer algorithms written in C into parallel codes having communication code in addition to the computation code segments which can be executed in message passing multicomputers.

A tool called SPRINT reported in (Cockx et al., 2007) automatically generates an executable concurrent SystemC model starting from a sequential code. While most of the reported techniques exploit data parallelism, SPRINT exploits functional parallelism in the behaviour. The tasks generated by SPRINT represent functional parallelism to yield tasks where each task implements a different subset of statements in the application. In contrast, data parallelism consists in executing the same code in parallel on different subsets of data. The work reported in (Girkar and Polychronopoulos, 1992) is focused on functional parallelism across loop and procedure boundaries. In their approach, a hierarchical task graph (HTG) is constructed from the initial sequential behaviour. The HTG provides a powerful intermediate representation mechanism which encapsulates parallelism of different types and scope levels, and is used for generation and optimization of parallel programs.

Turjan et al. (Turjan, 2007; Turjan et al., 2004) described an automatic transformation mechanism of nested-loop programs to Kahn process networks (KPNs). The main idea behind this approach is as follows. First, the computation carried out by a sequential application in a single process is divided into a number of separate computational processes. Secondly, some of the arrays used for data storage are transformed to dedicated FIFO buffers that are read using a *get* primitive and written into using *put* primitive, providing in this way a simple inter-process synchronization mechanism. They have implemented these transformation techniques in an embedded system design framework called Compaan (Kienhuis et al., 2000).

In (Ferrandi et al., 2007), an embedded system design tool chain for automatic generation of parallel code runnable on symmetric multiprocessor systems from an initial sequential specification is presented. The initial C specification is translated into an intermediate representation termed as a system dependence graph with feedback edges. Each loop body forms a partition of the graph. Statements with the same branch condition form control equivalent regions. Each control equivalent region forms a partition in the graph. The number of partitions are then optimized based on data and control dependencies. At the end, each partition (cluster of nodes) represents a different task.

An interactive tool support for parallelizing sequential applications was presented in (Ceng et al., 2008). In this work, an integrated framework called MAPS is presented which targets parallelizing C applications for MPSoC platforms. With embedded devices becoming more and more complex, an MPSoC programming framework should able to deal with multiple applications. With this objective, a framework to support multiple applications on top of the MAPS framework is presented in (Castrillon et al., 2010).

Running a parallel program which is obtained from a sequential behaviour on a multiprocessor architecture does not guarantee that the runtime requirements are met. Therefore, it may be necessary to further analyze and optimize the amount of concurrency in the parallel specification. It has been shown in (de Kock, 2002; Meijer et al., 2009, 2010b) that by changing the (interconnection) structure of a process network, the total execution time of an application can be improved. There are several kinds of transformations possible to manipulate the concurrency level of a parallel program. The process splitting transformation (Meijer et al., 2007, 2010a) selects a process from the original process network and creates a number of its copies such that the computational workload is distributed over these copies. The channel merging transformation reduces the number of communication channels (Turjan, 2007). Channel merging is relevant when the process network is to be mapped on a multiprocessor platform that consists of processing units, shared bus and shared memory. In such an architecture, communication and synchronization may be costly. So, merging channels may reduce the communication cost and the synchronization cost. Process clustering, described in (Davare et al., 2006), reduces the concurrency in the functional model by merging together functional processes. Transformations like message vectorization, computation migration and interprocess communication (IPC) selection have been discussed in (Fei et al., 2007). A complete design framework transforming an initial sequential behaviour to a concurrent behaviour and optimizing the latter before finally mapping it to a multiprocessor system is given in (Nikolov et al., 2008). The performance of the process merging transformation in embedded Multi-Processor Systems on Chip (MPSoCs) platforms is evaluated in (Meijer et al., 2010b). Similar work for process partitioning transformations is reported in (Meijer et al., 2009).

2.4.2 Verification of parallelizing transformations

An approach for symbolic model checking of process networks is introduced in (Strehl and Thiele, 2000). The authors have identified the problem with binary decision diagram based model checking of process networks and introduced a representation of multi valued functions called interval decision diagrams (IDDs). In this paper, interval diagram techniques are applied to symbolic model checking of process networks. The non-semantic preserving transformations of process networks during refinement steps of embedded system design is proposed in (Raudvere et al., 2008). Non-semantic-preserving transformations introduce lower level implementation details. In this approach, a set of verification properties for every non-semantic-preserving transformation are defined as CTL* formulae. The verification tasks are divided into two steps: (i) the local correctness of the non-semantic-preserving transformation is checked by preserving properties using model checking tool, and (ii) the global influence of the refinement to the entire system is studied through static analysis. In (Chen et al., 2003), the designs at deferent abstraction levels are automatically translated into Promela description and verified using SPIN model checker (Holzmann, 1997).

When a sequential behaviour is transformed into a parallel behaviour or a parallel behaviour is transformed into another parallel behaviour, it needs to ensure that (i) the transformed behaviour satisfies certain properties of the systems and (ii) it is functionally equivalent to the initial behaviour. For the first task, model checking is used as the primary verification approach for embedded system design. The verification models are automatically generated from the both input and transformed behaviours and the properties are checked using model checker like, SPIN (Holzmann, 1997), NuSMV (Cimatti et al., 2000), etc.. For the second task, model checking, however, cannot be used as it is more appropriate for property verification but not for behavioural verification.

For a complex system, like the embedded systems, however, a synthesizable specification is not restricted to some liveness or safety properties only; instead, it depicts behaviour at a high abstraction level in a procedural form. When a behaviour is transformed, it is important to show that the transformed behaviour preserves the functional behaviour of the initial specification. The current state of the art for embedded systems verification lacks in this aspect. As discussed in section 1.3, Kahn process network

(KPN) (Kahn, 1974) is often used to model parallel behaviours in the case of multimedia and signal processing applications,. In this work, we want to show the functional equivalence between a sequential behaviour and its correcponding KPN behaviour and also between two KPN behaviours.

2.5 Conclusion

In this chapter, we have discussed several applications of code motion transformations, loop transformations, arithmetic transformations, high-level to RTL transformations, RTL transformations, sequential to parallel transformations and parallel level transformations which are applied at various phases of contemporary embedded system design. Through the discussion, it is identified that those transformations are applied often during embedded system design. The state of the art verification methods for these transformations are also discussed in this chapter. In this process, we have identified some limitations of existing verification methods. In the subsequent chapters, we present verification methods for these transformations which overcome the limitations identified in this chapter.

Chapter 3

Verification of Code Motion Transformations

3.1 Introduction

Code motion techniques move operations from their original positions to some other places in order to reduce of the number of computations at run-time and to minimize the lifetimes of the temporary variables. As discussed in section 1.2, a code motion is said to be *non-uniform* when an operation is moved from a basic block (BB) preceding or succeeding a control block (CB) to only one of the conditional branches, or vice-versa. In contrast, a code motion is said to be *uniform* when an operation is moved to both the conditional branches after a CB from a preceding BB, or vice-versa. The input behaviour and the transformed behaviour are modelled as FSMDs in this work. In (Karfa, 2007), we have developed an FSMD based equivalence checking method which is capable of verifying uniform code motions. In this chapter, we have enhanced the method proposed in (Karfa, 2007) for verification of both uniform and non-uniform code motion transformations. To verify non-uniform code motions, we have identified that (i) certain data-flow properties must hold on the initial and the transformed behaviours for valid non-uniform code motions and (ii) the definition of equivalence of paths needs to be weakened. The present method automatically identifies the situations where such properties need to be checked during equivalence checking, generates the appropriate properties, and invokes the model checking tool

NuSMV (Cimatti et al., 2000) to verify them. The main contributions of the present work over (Karfa, 2007) are as follows: 1. Certain data-flow properties are identified to handle non-uniform code motions. Based on them, a weaker definition of equivalence of paths is given. 2. The equivalence theory of FSMDs is accordingly redefined based on the weaker definition of paths. 3. A completely automated method is developed for verification of both uniform and non-uniform code motions. 4. Correctness proof and complexity of the method are treated formally. 5. The developed method is used to verify the scheduling steps of a well known high-level synthesis tool SPARK (Gupta et al., 2003c), which applies a set of code transformations during scheduling.

This chapter is organized as follows. The FSMD models and the notion of equivalence of two FSMDs are introduced in section 3.2.1. A normalization technique of integer arithmetic expressions applied in our equivalence checking method to handle arithmetic transformations is also given in this section. Our verification scheme of non-uniform code motions are given in section 3.4. The enhancement of the method to handle multicycle and pipelined operations are discussed in section 3.5. The correctness and the complexity of the method are discussed in section 3.6. Experimental results are given in section 3.7. The paper is concluded in section 3.8.

3.2 Basic equivalence checking method

3.2.1 FSMDs and its paths

An FSMD (*finite state machine with datapath*) is a universal specification model (Gajski et al., 1992) that can represent all hardware designs. The FSMD is defined as an ordered tuple $\langle Q, q_0, I, V, O, f, h \rangle$, where

1. $Q = \{q_0, q_1, q_2, \ldots, q_n\}$ is the finite set of control states,
2. $q_0 \in Q$ is the reset state,
3. I is the set of primary input signals,
4. V is the set of storage variables, and Σ is the set of all data storage values or simply, data states,

5. O is the set of primary output signals,

6. $f : Q \times 2^{\mathcal{S}} \rightarrow Q$, is the state transition function and

7. $h : Q \times 2^{\mathcal{S}} \rightarrow U$, is the update function of the output and the storage variables, where S and U are as defined below.

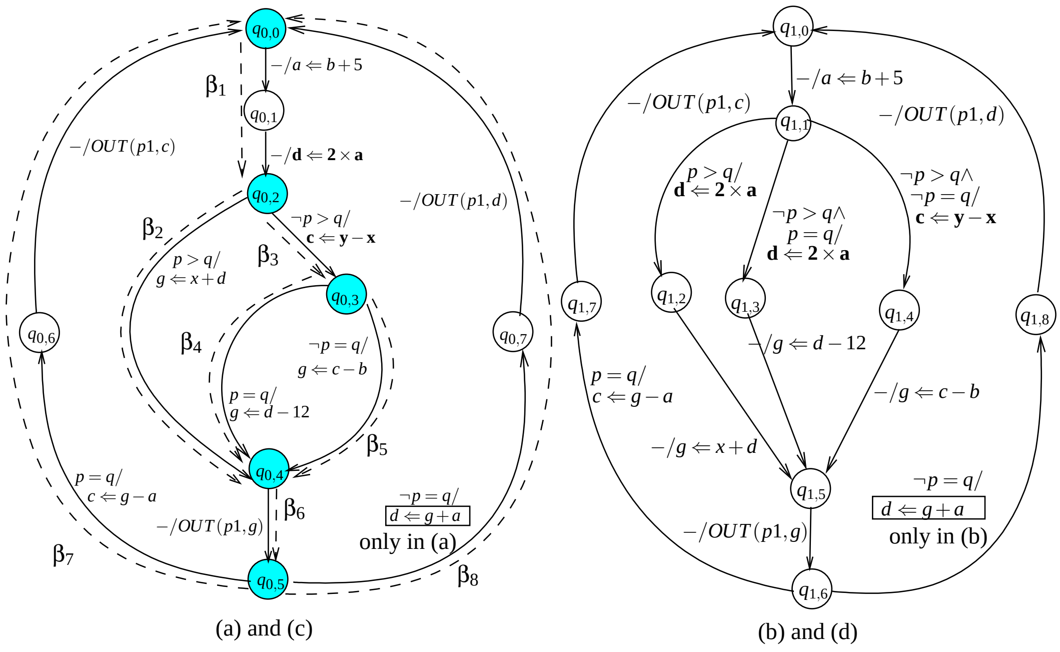
   (a) $\mathcal{S} = \{L \cup E\}$ is the set of status expressions where L is the set of Boolean literals of the form b or $\neg b$, $b \in B \subseteq V$ is a Boolean variable and E is the set of arithmetic predicates over $I \cup (V - B)$. Any arithmetic predicate is of the form $eR0$, where e is an arithmetic expression and $R \in \{==,\ \neq, >,\ \geq,\ <,\ \leq\}$.

   (b) U is a set of storage or output assignments of the form $\{x \Leftarrow e \mid x \in O \cup V$ and e is an arithmetic predicate or expression over $I \cup (V - B)\}$; it represents a set of storage or output assignments.

The implicit connective among the members of the set of status expressions which occurs as the second argument of the function f (or h) is conjunction. Parallel edges between two states capture disjunction of status expressions. The state transition function and the update function are such that the FSMD model remains deterministic. Thus, for any state $q \in Q$, if, $f(q, S_i)$, $1 \leq i \leq k$, are the only values defined, then $S_i \cap S_j = \phi$ for $i \neq j$, $1 \leq i, j \leq k$ and $S_1 \cup\ \ldots\ \cup S_k = true$. The same property holds for the update function h. It may be noted that we have not introduced any final state in the FSMD model as we assume that a system works in an infinite outer loop.

Example 1 The FSMD model M_0 for the behavioural specification given in figure 3.1(a) is as follows:

- $M_0 = \langle Q, q_0, I, V_0, O, f, h \rangle$, where
- $Q = \{q_{0,0},\ q_{0,1},\ q_{0,2},\ q_{0,3},\ q_{0,4},\ q_{0,5},\ q_{0,6},\ q_{0,7}\}$, $q_0 = q_{0,0}$, $V_0 = \{a,\ b,\ c,\ d,\ g,\ p,\ q,\ x,\ y\}$, $I = \{b,\ p,\ q, x,\ y\}$, $O = \{p1\}$,
- $U = \{a \Leftarrow b + 5,\ d \Leftarrow 2 \times a,\ c \Leftarrow y - x,\ g \Leftarrow x + d,\ g \Leftarrow d - 12,\ g \Leftarrow c - b,\ c \Leftarrow g - a,\ d \Leftarrow g + a\}$,
- $\mathcal{S} = \{p > q,\ p = q\}$,



$q_{0,0}$
$-/a \Leftarrow b+5$
$q_{0,1}$
$-/\mathbf{d} \Leftarrow \mathbf{2} \times \mathbf{a}$
$q_{0,2}$
$\neg p > q/$ $\mathbf{c} \Leftarrow \mathbf{y} - \mathbf{x}$
$p > q/$ $g \Leftarrow x+d$
$q_{0,3}$
$\neg p = q/$ $g \Leftarrow c-b$
$p = q/$ $g \Leftarrow d-12$
$q_{0,4}$
$-/OUT(p1,g)$
$q_{0,5}$
$p = q/$ $c \Leftarrow g-a$
$q_{0,6}$
$-/OUT(p1,c)$
$q_{0,7}$
$-/OUT(p1,d)$
$\neg p = q/$ $\boxed{d \Leftarrow g+a}$ only in (a)
β_1 β_2 β_3 β_4 β_5 β_6 β_7 β_8
(a) and (c)
$q_{1,0}$
$-/a \Leftarrow b+5$
$q_{1,1}$
$-/OUT(p1,c)$
$-/OUT(p1,d)$
$p > q/$ $\mathbf{d} \Leftarrow \mathbf{2} \times \mathbf{a}$
$\neg p > q \wedge$ $p = q/$ $\mathbf{d} \Leftarrow \mathbf{2} \times \mathbf{a}$
$\neg p > q \wedge$ $\neg p = q/$ $\mathbf{c} \Leftarrow \mathbf{y} - \mathbf{x}$
$q_{1,7}$ $q_{1,2}$ $q_{1,3}$ $q_{1,4}$ $q_{1,8}$
$-/g \Leftarrow d-12$
$p = q/$ $c \Leftarrow g-a$
$-/g \Leftarrow c-b$
$-/g \Leftarrow x+d$
$q_{1,5}$
$\neg p = q/$ $\boxed{d \Leftarrow g+a}$ only in (b)
$-/OUT(p1,g)$
$q_{1,6}$
(b) and (d)

Figure 3.1: Example of reverse speculation: (a) and (c): Two FSMDs are shown combined representing two behaviours before scheduling; (b) and (d): Two FSMDs are shown combined representing the behaviours after scheduling of the FSMD in (a) and in (c), respectively

- f and h are as defined in the transition graph shown in figure 3.1(a). Some typical values of f and h are as follows:
  - $f(q_{0,0}, \{true\}) = q_{0,1}$, $f(q_{0,5}, \{p = q2\}) = q_{0,6}$,
  - $h(q_{0,5}, \{p = q\}) = \{c \Leftarrow g - a\}$, $h(q_{0,4}, \{true\}) = \{OUT(p1, g)\}$.

□

A (finite) *path* α from q_i to q_j, where $q_i, q_j \in Q$, is a finite transition sequence of states of the form $\langle q_i = q_1 \underset{\overrightarrow{c_1}}{} q_2 \underset{\overrightarrow{c_2}}{} \cdots \underset{\overrightarrow{c_{n-1}}}{} q_n = q_j \rangle$ such that $\forall l, 1 \leq l \leq n-1, \exists c_l \in 2^S$ such that $f(q_l, c_l) = q_{l+1}$, and q_k, $1 \leq k \leq n-1$, are all distinct. Only the last state q_n may be identical to any q_k, $1 \leq k \leq n-1$. In contrast, a *finite walk* is also such a finite transition sequence of states where any state can repeat. The *condition of execution of the path* α, denoted as R_α, is a logical expression over the variables in V and the inputs I such that R_α is satisfied by the initial data state of the path[1] iff the path α is traversed. Thus, R_α is the weakest precondition of the path α (Gries, 1987).

[1]Data state of a variable at some control point is its value in that control point

The *data transformation of a path* α over V, denoted as r_α, is the tuple $\langle s_\alpha, O_\alpha \rangle$; the first member s_α, termed as storage (variable) transformation of α, is an ordered tuple $\langle e_i \rangle$ of algebraic expressions over the variables in V and the inputs in I such that the expression e_i represents the value of the variable v_i after the execution of the path in terms of the initial data state of the path; the second member $O_\alpha = [OUT(P_{i_1}, e_1), OUT(P_{i_2}, e_2), \ldots]$ represents the output list along the path α. More specifically, for every expression e output to port P along the path α, there is a member $OUT(P, e)$ in the list, appearing in the order in which the outputs occur in α. We have to maintain the list for outputs along a path; otherwise, all the outputs other than the last one will be missed in the data transformations if there are more than one output through a single port within a path. The condition R_α and the data transformations r_α can be obtained by backward substitution or by forward substitution (Manna, 1974). Often, for brevity, the above path α is represented as $[q_1 \Rightarrow q_n]$.

Example 2 Let us consider the path $\alpha = q_{0,0} \rightarrow q_{0,1} \rightarrow q_{0,2} \xrightarrow[\neg p > q]{} q_{0,3} \xrightarrow[p=q]{} q_{0,4} \rightarrow q_{0,5} \xrightarrow[p=q]{} q_{0,6} \rightarrow q_{0,0}$ in the FSMD in figure 3.1(a). The computation of $[R_\alpha,\ r_\alpha]$ for this path by forward substitution method is as follows:

At $q_{0,0}$: $[true,\ \langle\langle a,\ d,\ g,\ c\rangle, -\rangle]$.

At $q_{0,1}$: $[true,\ \langle\langle b+5,\ d,\ g,\ c\rangle, -\rangle]$.

At $q_{0,2}$: $[true,\ \langle\langle b+5,\ 2\times(b+5),\ g,\ c\rangle, -\rangle]$.

At $q_{0,3}$: $[\neg p > q,\ \langle\langle b+5,\ 2\times(b+5),\ g,\ y-x\rangle, -\rangle]$.

At $q_{0,4}$: $[\neg p > q \wedge p = q,\ \langle\langle b+5,\ 2\times(b+5),\ 2\times(b+5)-12,\ y-x\rangle, -\rangle]$.

At $q_{0,5}$: $[\neg p > q \wedge p = q,\ \langle\langle b+5,\ 2\times(b+5),\ 2\times(b+5)-12,\ y-x\rangle, OUT(p1_1, 2\times(b+5)-12)\rangle]$.

At $q_{0,6}$: $[\neg p > q \wedge p = q \wedge p = q,\ \langle\langle b+5,\ 2\times(b+5),\ 2\times(b+5)-12,\ 2\times(b+5)-12-(b+5)\rangle, OUT(p1_1, 2\times(b+5)-12)\rangle]$.

At $q_{0,0}$: $[\neg p > q \wedge p = q \wedge p = q,\ \langle\langle b+5,\ 2\times(b+5),\ 2\times(b+5)-12,\ 2\times(b+5)-12-(b+5)\rangle, OUT(p1_1, 2\times(b+5)-12), OUT(p1_2, 2\times(b+5)-12-(b+5))\rangle]$.

The expressions in $[R_\alpha,\ r_\alpha]$ can be simplified. After simplification,

$R_\alpha = \neg p > q \wedge p = q$ and

$r_\alpha = \langle\langle b+5,\ 2\times b+10,\ 2\times b-2,\ b-7\rangle,\ OUT(p1_1, 2\times b-2),\ OUT(p1_2, b-7)\rangle.$

During checking the equivalence of two arithmetic expressions, such simplifications are sometimes essential because the scheduler may itself have simplified some arithmetic expressions of the initial behaviour (Gupta et al., 2000, 2002; Landwehr and Marwedel, 1997; Potkonjak et al., 1993; Zory and Coelho, 1998). Accordingly, a normalization method and some simplification rules of arithmetic expressions are incorporated in this work. The normalization method and the simplification rules are presented in the next subsection. □

The characteristic formula $\tau_\alpha(\bar{v},\bar{v}_f,O)$ of the path α is $R_\alpha(\bar{v}) \wedge (\bar{v}_f = s_\alpha(\bar{v})) \wedge (O = O_\alpha(\bar{v}))$, where s_α is the data transformation and O_α is the output list in the path α, $\bar{v}$ represents a vector of variables of $I \cup V$, $\bar{v}_f$ represents a vector of variables of V. The formula captures the following: if the condition of execution R_α of the path α is satisfied by the (initial) vector $\bar{v}$ at the beginning of the path, then the path is executed and after execution, the final vector $\bar{v}_f$ of variable values becomes $s_\alpha(\bar{v})$ and the output $O_\alpha(\bar{v})$ is produced. Let $\tau_\alpha(\bar{v},\bar{v}_f,O) : R_\alpha(\bar{v}) \wedge (\bar{v}_f = s_\alpha(\bar{v})) \wedge (O = O_\alpha(\bar{v}))$ be the characteristic formula of the path α and $\tau_\beta(\bar{v},\bar{v}_f,O) : R_\beta(\bar{v}) \wedge (\bar{v}_f = s_\beta(\bar{v})) \wedge (O = O_\beta(\bar{v}))$ be the characteristic formula of the path β. The characteristic formula for the concatenated path $\alpha\beta$ is

$$\tau_{\alpha\beta}(\bar{v},\bar{v}_f,O) = \exists\bar{v}_\alpha \exists O_1 \exists O_2 (\tau_\alpha(\bar{v},\bar{v}_\alpha,O_1) \wedge \tau_\beta(\bar{v}_\alpha,\bar{v}_f,O_2))$$

$$= R_\alpha(\bar{v}) \wedge R_\beta(s_\alpha(\bar{v})) \wedge (\bar{v}_f = s_\beta(s_\alpha(\bar{v}))) \wedge\ (O = O_\alpha(\bar{v})O_\beta(s_\alpha(\bar{v}))).$$

The list O is the concatenated output list of $O_\alpha(\bar{v})$ and $O_\beta(s_\alpha(\bar{v}))$.

3.2.2 Normalization of arithmetic expressions

Specification for embedded system implementing algorithmic computation over integers involve the whole of integer arithmetic which is undecidable; thus, a canonical form does not exist for integer arithmetic. Instead, in this work, we use the following normal forms of expressions, adapted from (Sarkar and De Sarkar, 1989) for both integer and real expressions. The normalization process reduces many computationally equivalent formulas syntactically identical as it forces all the formulas to follow a uniform structure (Sarkar and De Sarkar, 1989). In the following, the normal form chosen for the formulas and the simplification carried out on the normal form during the normalization phase are described in brief.

The data transformation of a path is an ordered tuple $\langle e_i \rangle$ of arithmetic expressions such that the expression e_i represents the value of the variable v_i after the execution of the path in terms of the initial data state. Each arithmetic expression in data transformation can be represented in the *normalized sum form*. A normalized sum (S) is a sum of terms with at least one constant term; each term (T) is a product of primaries with a non-zero constant primary; each primary (P) is a storage variable, an input variable or of the form $abs(s)$, $mod(s_1, s_2)$ or $div(s_1, s_2)$, where s, s_1, and s_2 are normalized sums. In addition to the above structure, any normalized sum is arranged by a lexicographic ordering of its constituent subexpressions from the bottom-most level, i.e., from the level of simple primaries. This syntactic entities are defined by means of productions of the following grammar.

Definition 1 *1.* $S \rightarrow S + T \mid c_s$*, where* c_s *is an integer.*

2. $T \rightarrow T * P \mid c_t$*, where* c_t *is an integer.*

3. $P \rightarrow abs\ (S) \mid (S)\ mod\ (S) \mid S \div C_d\ |v|\ c_m$*, where* $v \in I \cup V$ *and* c_m *is an integer,*

4. $C_d \rightarrow S \div C_d \mid S$*.*

A condition of execution of a path is an arithmetic relation of the form $S\,R\,0$, where S is a normalized sum and $R \in \{\leq, \geq, >, <, =, \neq\}$. The relation $>$ ($<$) can be reduced to $\geq$ ($\leq$) over integers. For example, $x - y > 0$ can be reduced to $x - (y - 1) >= 0$. Negated relational literals are suitably modified to absorb the negation.

In addition, the following simplifications are carried out for integer expressions. (i) The common subexpressions in a sum are collected. Thus, $x^2 + 3x + 4z + 7x$ is simplified to $x^2 + 10x + 4z + 0$. (ii) A relational literal is reduced by a common constant factor, if any, and the literal is accordingly simplified. For example, $3x^2 + 9xy + 6z + 7 \geq 0$ is simplified to $x^2 + 3xy + 2z + 2 \geq 0$, where $\lfloor 7/3 \rfloor = 2$.

The following simplification is carried out for both real and integer expressions. (iii) Literals are deleted from a conjunction by the rule "if $(l \Rightarrow l')$ then $l \wedge l' \equiv l$." (iv) If $l \Rightarrow \neg l'$, then a conjunction having literals l and l' is reduced to false. Implication of a relational literal by another is identified by the method described in (Sarkar and De Sarkar, 1989). Associativity and commutativity of $+$, $-$, $*$, distributivity of $*$ over $+$, $-$, symmetry of $\{=, \neq\}$, reflexivity of $\{\leq, \geq, =\}$ and Irreflexivity of $\{\neq, <, >\}$ are accounted for by the above transformations.

Let us revisit the example 2. In this example, we simplify the arithmetic expressions in the data transformations and the arithmetic predicates in the condition of execution of a path. For example, expression $2 \times (b+5) - 12 - (b+5)$ is simplified to $b - 7$. The normalized sum form of the expression $2 \times (b+5) - 12 - (b+5)$ is $2 \times b + (-1) \times b + (-7)$. The rule (i) helps us in reaching the simplified form also. Similarly, the condition of execution of that path $\neg p > q \wedge p = q \wedge p = q$ is simplified to $p = q$ by applying rule (iii). The above described normalization form along with the simplification rules help us reach such simplified form. Let us consider another possible path $\beta = q_{0,3} \xrightarrow[p=q]{} q_{0,4} \rightarrow q_{0,5} \xrightarrow[\neg p=q]{} q_{0,7} \rightarrow q_{0,0}$, where p and q do not change in the transition $q_{0,4} \rightarrow q_{0,5}$. For this path, $R_\beta : p = q \wedge \neg p = q$. So, R_β becomes false after application of rule (iv). Actually, the path β is not a valid path since it never executes in practice. So, our normalization procedure and the simplification rules enable us ignore such paths during equivalence checking.

3.3 Equivalence problem formulation

Let the input behaviour be represented by the FSMD $M_0 = \langle Q_0, q_{0,0}, I, V_0, O, f_0, h_0 \rangle$ and the transformed behaviour be represented by the FSMD $M_1 = \langle Q_1, q_{1,0}, I, V_1, O, f_1, h_1 \rangle$. Our main goal is to verify whether M_0 behaves exactly as M_1. This means that for all possible input sequences[2], the execution traces of M_0 and M_1 produce the same sequence of outputs on each output port. A *computation* of an FSMD is a finite walk

[2]Note that unlike output sequences, we need not to keep track of the input sequences because every input redefines a variable (like any assignment statement).

from the reset state back to itself without having any intermediary occurrence of the reset state. So, a computation, which represents one possible execution of an FSMD, takes an input sequence and produces an output sequence. The equivalence of computations can be defined as follows.

Definition 2 (Equivalence of Computations) *A computation c_1 of an FSMD M_0 is equivalent to a computation c_2 of another FSMD M_1 if $R_{c_1} \equiv R_{c_2}$ and $O_{c_1} \equiv O_{c_2}$, where R_{c_1} and R_{c_2} represent the conditions of execution of c_1 and c_2, respectively and O_{c_1} and O_{c_2} are the output lists of c_1 and c_2, respectively. The fact that the computation c_1 is equivalent to the computation c_2 is denoted as $c_1 \simeq c_2$.*

Similarly, a path β is said to be equivalent to a path α iff $R_\beta \simeq R_\alpha$ and $r_\beta \simeq r_\alpha$. The fact that a path β is computationally equivalent to a path α is denoted as $\beta \simeq \alpha$.

The following two definitions capture the notion of equivalence of FSMDs.

Definition 3 (Containment of FSMDs) *An FSMD M_0 is said to be contained in an FSMD M_1, symbolically $M_0 \sqsubseteq M_1$, if for any computation c_0 of M_0, there exists a computation c_1 of M_1 such that $c_0 \simeq c_1$.*

Definition 4 (Equivalence of FSMDs) *Two FSMDs M_0 and M_1 are said to be computationally equivalent, if $M_0 \sqsubseteq M_1$ and $M_1 \sqsubseteq M_0$.*

3.3.1 Path cover and equivalence of FSMDs

An FSMD may consist of an infinite number of computations. However, any computation μ of an FSMD M can be looked upon as a computation along some concatenated path $[\alpha_1\alpha_2\alpha_3 ... \alpha_k]$ of M such that the path α_1 emanates from and the path α_k terminates in the reset state q_0 of M, for $1 \leq i < k$, α_i terminates in the initial state of the path α_{i+1} and α_i's may not all be distinct. Hence, we have the following definition.

Definition 5 (Path cover of an FSMD) *A finite set of paths $P = \{p_0, p_1, p_2, \ldots, p_k\}$ is said to be a path cover of an FSMD M if any computation c of M can be looked upon as a concatenation of paths from P.*

The following theorem can be concluded from definitions 3 and 5 (Karfa, 2007).

Theorem 1 *An FSMD M_0 is contained in another FSMD M_1 ($M_0 \sqsubseteq M_1$), if there exists a finite cover $P_0 = \{p_{00}, p_{01}, \ldots, p_{0l}\}$ of M_0 for which there exists a set $P_1 = \{p_{10},\ p_{11}, \ldots,\ p_{1l}\}$ of paths of M_1 such that $p_{0i} \simeq p_{1i}$, $0 \leq i \leq l$.*

The statement of theorem 1 suggests the following steps of a verification method for checking equivalence of two FSMDs:

1. Construct the set P_0 of paths of M_0 so that P_0 covers M_0. Let $P_0 = \{p_{0,0}, p_{0,1}, \cdots, p_{0,k}\}$.
2. Show that $\forall p_{0,i} \in P_0, there\ exists\ a\ path\ p_{1,j}$ of M_1 such that $p_{0,i} \simeq p_{1,j}$.
3. Repeat steps 1 and 2 with M_0 and M_1 interchanged.

The question remains how one can find the path cover of an FSMD. It is difficult to find a path cover of the whole computation comprising only finite paths because of loops. Hence, any computation is split into paths by putting *cutpoints* at various places in the FSMD so that each loop is cut in at least one cutpoint. The set of *all paths from a cutpoint to another cutpoint without having any intermediary cutpoint* is a path cover of the FSMD (Floyd, 1967; Hoare, 1969; King, 1980). In this work, the starting cutpoints are defined in any FSMD as follows:

1. The reset state is chosen.
2. A state q_i is chosen, if there is a divergence or convergence of flow from q_i.

The cutpoints chosen by the above rules cut each loop of the FSMD in at least one cutpoint, because each internal loop has an exit point.

3.3.2 A method to handle uniform code motions

The control structure of the input behaviour may be modified by the path based scheduler and also some operations in the behaviour may be moved beyond the conditional

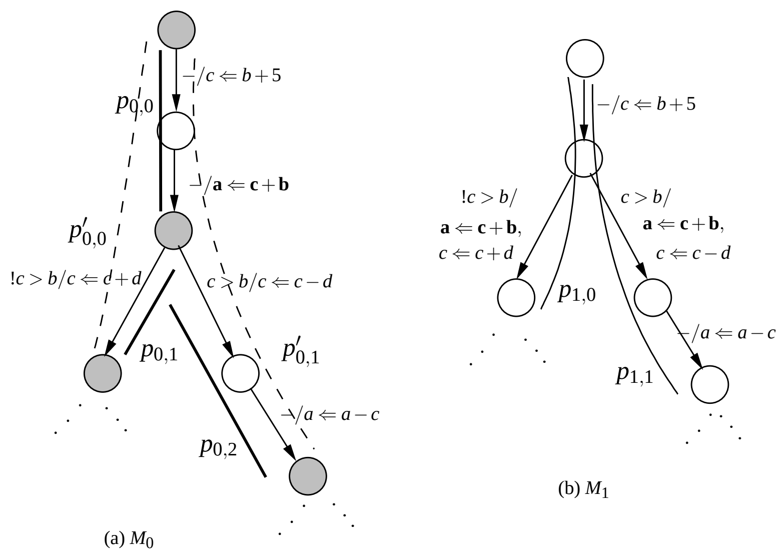


Figure 3.2: An example of duplicating down

statements by application of code motion techniques. Therefore, the basic equivalence checking method described in subsection 3.3.1 fails to find equivalent paths for some members of the path cover P_0 for the above choices of cutpoints. One situation of code motion is explained with figure 3.2. In this behaviour, the operation $a \Leftarrow c+b$ is duplicated down to both the branches in the transformed behaviour. The cutpoints are shaded in the figure. The paths of the initial path cover of M_0 are shown as bold lines. In particular, the path cover P_0 is $\{p_{0,0}, p_{0,1}, p_{0,2}\}$. The basic equivalence checking method fails to find the equivalent of path $p_{0,0}$ in M_1 because of mismatch in the transformation of the variable a due to code motion. The idea of path extension was proposed in (Karfa, 2007) for handling control structure modification. The mechanism, however, also alleviates the problem of mismatch of variables values due to uniform code motions. The works as follows. As the equivalent of $p_{0,0}$ cannot be found in M_1, the method extends this path in all possible ways up to the next set of cutpoints and checks the equivalence of every extended path in M_1. The method, however, does not extend a path beyond loops. In this way, the extended paths of $p_{0,0}$ are $p'_{0,0}$ and $p'_{0,1}$ (shown as dashed lines in the figure). It might be noted that the equivalent paths of $p'_{0,0}$ and $p'_{0,1}$ can be found as $p_{1,0}$ and $p_{1,1}$, respectively.

This path extension based method works fine if consecutive path segments are merged as in the case of a path based scheduler. It also works well for the following cases of uniform code motions: (i) a code segment before a conditional block has been moved to all the branches, (ii) a code segment after merging of branches has been moved to all the branches before the merge point, (iii) a code segment before a control block has been moved after merging takes place following the control block.

The variable set of the initial and the transformed FSMD may not be exactly the same. It is because of the fact that some variables may be removed or some variables may get introduced due to code motion and arithmetic transformations. In (Karfa, 2007), therefore, the equivalence of paths are defined only on the variables in $V_0 \cap V_1$. In other words, two paths α and β are equivalent iff R_α, r_α, R_β and r_β are defined over $V_0 \cap V_1$ and $R_\alpha = R_\beta$ and $r_\alpha = r_\beta$. Any expression (arithmetic or status) is *defined* over the variable set $V_0 \cap V_1$ if all the variables it involves belong to $V_0 \cap V_1$. Therefore, if R_α, r_α, R_β and r_β become undefined (when they involve a variable which do not belong to $V_0 \cap V_1$), the method in (Karfa, 2007) reports a possible non-equivalence of paths.

3.4 Verification of non-uniform code motions

In the present section, we introduce non-uniform code motion through an example and then illustrate how the basic method described so far fails to handle such code motions; the exercise helps in identifying the features along which enhancement is needed. The method is then enhanced accordingly.

3.4.1 An example of non-uniform code motion

Let us consider the FSMDs in figure 3.1. The transformed behaviour in figure 3.1(b) is obtained from the input behaviour shown in figure 3.1(a) by application of non-uniform code motions. Specifically, the operation $d \Leftarrow 2 \times a$ associated with the transition $q_{0,1} \rightarrow q_{0,2}$ of the original behaviour in the FSMD M_0 of figure 3.1(a) is moved to the transitions $q_{1,1} \rightarrow q_{1,2}$ and $q_{1,1} \rightarrow q_{1,3}$ of the FSMD M_1 of figure 3.1(b). The operation, however, is not moved to the other branch $\langle q_{1,1} \xrightarrow[\neg p > q \wedge \neg p = q]{} q_{1,4} \rangle$ from the state $q_{1,1}$. Similarly, the other operation $c \Leftarrow y - x$ associated with the transition $q_{0,2} \rightarrow q_{0,3}$ of the original behaviour is moved only to the transition $q_{1,1} \rightarrow q_{1,4}$ from the state $q_{1,1}$. So, these are instances of non-uniform duplicating down code motion. It may be noted that these code motions reduce the execution time of the behaviour and the number of computations by one unit when $\neg p > q$ is true. They also reduce the lifetimes of the variables d and c by one unit. The first transformation (of moving $d \Leftarrow 2 \times a$) is permissible because the branch $q_{0,3} \xrightarrow[\neg p = q]{} q_{0,4}$ and the rest of the code staring from the state $q_{0,4}$ in FSMD M_0 do not use this definition $(2 \times a)$ of d. Similar observations hold for the operation $c \Leftarrow y - x$ also. Therefore, these two FSMDs produce the same outputs for all possible inputs; hence they are equivalent.

Let us now consider the FSMD in figure 3.1(c) (the only difference between the FSMDs in figures 3.1(a) and 3.1(c) is that the operation $d \Leftarrow g + a$ is present in the transition $q_{0,5} \xrightarrow[\neg p = q]{} q_{0,7}$ of M_0 of figure 3.1(a) but not present in the same transition of figure 3.1(c)). Let the same code motion described above be applied on this modified FSMD. The corresponding transformed FSMD M_1 is shown in figure 3.1(d) (again the only difference between the FSMDs in figure 3.1(b) and 3.1(d) is that the operation $d \Leftarrow g + a$ is present in the transition $q_{1,6} \xrightarrow[\neg p = q]{} q_{1,8}$ of M_1 of figure 3.1(b) but not present in the same transition of figure 3.1(d)). In this case, the non-uniform movement of the operation $d \Leftarrow 2 \times a$ is not correct because it leads to a non-equivalent execution of the FSMDs. Specifically, the output value of d at port $p1$ is $2 \times a$ in the execution $c1 = q_{0,0} \rightarrow q_{0,1} \rightarrow q_{0.2} \xrightarrow[\neg p > q]{} q_{0,3} \xrightarrow[\neg p = q]{} q_{0,4} \rightarrow q_{0,5} \xrightarrow[\neg p = q]{} q_{07} \rightarrow q_{0,0}$ of the FSMD M_0 whereas the output value of d at the same port $p1$ is undefined in the execution $c2 = q_{1,0} \rightarrow q_{1,1} \xrightarrow[\neg p > q \wedge \neg p = q]{} q_{1,4} \rightarrow q_{1,5} \rightarrow q_{1,6} \xrightarrow[\neg p = q]{} q_{1,8} \rightarrow q_{1,0}$ of FSMD M_1 in figure 3.1(d). It may be noted that $c2$ is the only execution in M_1 which can be equivalent to $c1$ because their execution conditions match and the FSMDs being deterministic no other path from $q_{1,0}$ can have the same condition of execution. The objective of this work is to develop an equivalence checking method which can establish such equivalence or non-equivalence of the FSMDs. Moreover, the method should be as efficient as possible.

3.4.2 A scheme for verifying non-uniform code motions

As discussed above, the method given in (Karfa, 2007) extends a path in all possible ways when the method fails to find its equivalent path. In the case of non-uniform code motions, however, this approach confronts some problem. Let us consider, for example, the following situation depicted in figure 3.3. Let $p_{0,1} = q_{0,i} \Rightarrow q_{0,f}$ be a path segment (in M_0) having a code c transforming a variable v. Let the code c be moved (non-uniformly) in M_1 to one path segment $p_{1,2} = q_{1,f} \Rightarrow q'_{1,f}$ emanating from

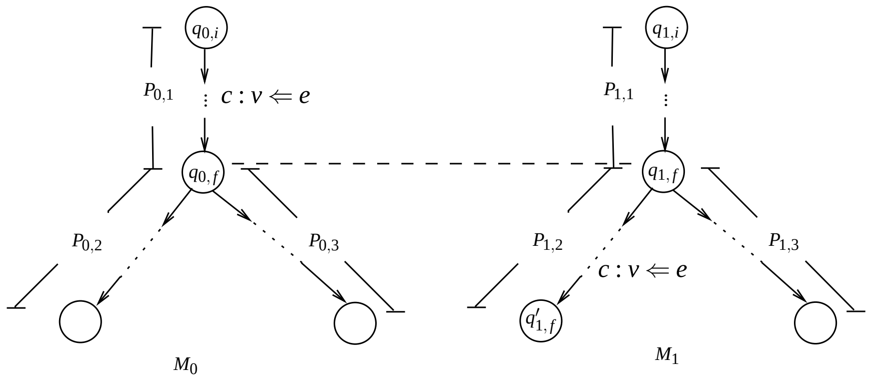


Figure 3.3: An instance of non-uniform code motion

the corresponding state $q_{1,f}$ of $q_{0,f}$ and not moved to another path segment $p_{1,3}$ also emanating from $q_{1,f}$. For the concatenated path segment, $p_{0,1}p_{0,2}$ in M_0 obtained by extending $p_{0,1}$ incorporating the path segment $p_{0,2}$, we will have an equivalent path $p_{1,1}p_{1,2}$ but for the path $p_{0,1}p_{0,3}$ obtained by extending $p_{0,1}$ incorporating the path segment $p_{0,3}$, the equivalent path will be elusive. However, it may be the case that in $p_{0,3}$ and in the rest of the behaviour from the end state of $p_{0,3}$, the property that the variable *v is not used at all or may have been defined before used* is true (equivalently, *v is used before being defined* is false). In such situations, the path $p_{1,1}p_{1,3}$ is actually the equivalent path of $p_{0,1}p_{0,3}$. Extending $p_{0,1}p_{0,3}$ (as is done by the method given in (Karfa, 2007)), however, never reveals that equivalence. Following example ilusutraes the fact further.

Example 3 Let us consider the path $\beta = q_{0,0} \rightarrow q_{0,1} \rightarrow q_{0,2} \xrightarrow[\neg p>q]{} q_{0,3} \xrightarrow[\neg p=q]{} q_{0,4}$ of the FSMD given in figure 3.1(a). For this path β, $R_\beta = \neg p > q \wedge \neg p = q$ and $r_\beta = \langle\langle b+5,\ 2\times b+10,\ y-x-b,,\ y-x\rangle, -\rangle$, where the order of the variables are $\langle a,\ d,\ g,\ c\rangle$. Let us also consider the path $\alpha = q_{1,0} \rightarrow q_{1,1} \xrightarrow[\neg p>q \wedge \neg p=q]{} q_{1,4} \rightarrow q_{1,5}$ of the FSMD given in figure 3.1(b). The condition of execution of α is $\neg p > q \wedge \neg p = q$ and the data transformation of α $r_\alpha = \langle\langle b+5,\ d,\ y-x-b,\ y-x\rangle, -\rangle$. It may be noted that the conditions of execution of both β and α are the same but the data transformations of d ($\in V_0 \cap V_1$) are not the same in r_β and r_α. Hence, β and α are not equivalent. According to the method presented in (Karfa, 2007), the path β needs to be extended. It can be shown that after extending β twice, neither the equivalence of one of the extended paths $q_{0,0} \rightarrow q_{0,1} \rightarrow q_{0,2} \xrightarrow[\neg p>q]{} q_{0,3} \xrightarrow[\neg p=q]{} q_{0,4} \rightarrow q_{0,5} \xrightarrow[p=q]{} q_{0,6} \rightarrow q_{0,0}$ can be found in the FSMD in figure 3.1(b) nor the path can be further extended. Therefore, the method in

(Karfa, 2007) reports a possible non-equivalence between these two FSMDs.

We, however, note that the definition of d in path β is redundant because the defined value has no further use. We can conclude that if a variable v (in $V_0 \cap V_1$) is transformed in a path but the value acquired thereof is not used in the rest of the behaviour, then the transformation of v in that path can be ignored since it has no further use. Accordingly, α can be shown to be the equivalent path of β. $\square$

There is another shortcoming of the method given in (Karfa, 2007); it establishes the equivalence of paths ignoring the transformations of the variables which are not in $V_0 \cap V_1$. This approach, however, leads to false-negative results in the case of non-uniform code motion. It is because of the fact that the condition of execution R_α or the data transformation r_α of a path α is defined in terms of the values of the variables at the start state of the path; these values, in turn, may depend on the uncommon variables. Instead of setting the equivalence of paths by blindly ignoring the transformations of the variables which are not in $V_0 \cap V_1$, a selective path extension accommodating propagation of values of the variables may actually reveal the equivalence of FSMDs in the case of non-uniform code motion. This situation is illustrated with the help of the following example.

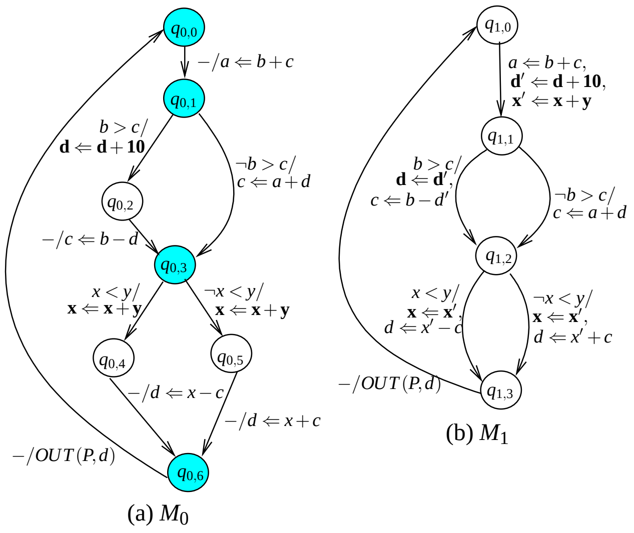


Figure 3.4: An example of speculative code motion

Example 4 Let us consider the FSMDs M_0 and M_1 in figure 3.4. Let us assume for simplicity that in the states $q_{0,0}$ and $q_{1,0}$ each variable has the same initial value. Here, the operation $d \Leftarrow d + 10$ associated with the branch $q_{0,1} \xrightarrow[b>c]{} q_{0,2}$ of the FSMD M_0 in figure 3.4(a) has been speculated out to the transition $q_{1,0} \rightarrow q_{1,1}$ in the FSMD M_1 in figure 3.4(b) and the result is stored in the variable d'. Similarly, the operation $x \Leftarrow x + y$, which occurs in both the branches from the state $q_{0,3}$ in M_0, is speculated out to the transition $q_{1,0} \rightarrow q_{1,1}$ in the FSMD M_1 and the result is stored in the variable x'.

The cutpoints are shaded in the FSMD in figure 3.4(a). The variables d' and x' are updated in the path $\alpha = q_{1,0} \Rightarrow q_{1,1}$ but they do not belong to $V_0 \cap V_1$. So, the method given in (Karfa, 2007) ascertains $\beta = q_{0,0} \Rightarrow q_{0,1}$ as the equivalent path of α ignoring d' and x'; this, however, leads to a wrong inference (a false negative) subsequently as explained below. While dealing with the path $q_{0,1} \xrightarrow[b>c]{} q_{0,3}$, it is found that the final value of d does not match with the same in the potential equivalent path (i.e., $q_{1,1} \xrightarrow[b>c]{} q_{1,2}$). So, the method given in (Karfa, 2007) reports a possible non-equivalence of these two FSMDs (which is not true in this case).

Instead of inferring (greedily) that $q_{0,0} \Rightarrow q_{0,1}$ and $q_{1,0} \Rightarrow q_{1,1}$ are equivalent, it should be examined whether the values of d' or x' are used subsequently after the state $q_{1,1}$; if so, it is required to extend the path $\beta = q_{0,0} \Rightarrow q_{0,1}$ instead of saying it is equivalent with $\alpha = q_{1,0} \Rightarrow q_{1,1}$. On the other hand, if the property *"used before being defined"* is false in the rest of the code starting from $q_{1,1}$ for both d' and x', then we can infer that α is the equivalent of β. In case of the path $q_{1,0} \Rightarrow q_{1,1}$ (which has the potential of being equivalent of the path $q_{0,0} \Rightarrow q_{0,1}$), the property *"used before being defined"* becomes true for both d' and x' at $q_{1,1}$. So, the original path $q_{0,0} \Rightarrow q_{0,1}$ will be extended. The extended paths are $q_{0,0} \rightarrow q_{0,1} \xrightarrow[b>c]{} q_{0,2} \rightarrow q_{0,3}$ and $q_{0,0} \rightarrow q_{0,1} \xrightarrow[\neg b>c]{} q_{0,3}$. It may be noted that the property *"used before being defined"* is false for the variable d' at $q_{1,2}$. So, we can now ignore the value of d'. However, the extended paths need to be extended again as the property is still true for x' at $q_{1,2}$. The reverse of the situation, i.e., some variables are transformed in β in M_0 but not in α, can also happen. In that case, we have to check the same property at the end state of β. □

The examples given in this subsection, therefore, reveal the following: (i) Example 3 reveals a situation where checking equivalence of all the variables in $V_0 \cap V_1$ (for equivalence of paths) may result finally in false-negative result through some unnecessary path extensions. (ii) Example 4 reveals a situation where ignoring blindly the variables which are not in $V_0 \cap V_1$ may result in a false negative inference. In both the cases, we have identified that checking equality of only those variables which are used later without further definition suffice for checking equivalence of paths. The property "*used before being defined*" helps us in both cases to find those variables. Therefore, we can have a weaker definition of equivalence of paths where, besides showing the equivalence of the respective conditions of execution and the respective output lists, it is sufficient to show the equivalence of the variable transformations in s_α of *only those variables whose values are used later*. In the subsequent subsection, we shall formally define strong and weak equivalence of paths and then give the proof of theorem 1 accommodating weak equivalence of paths.

3.4.3 Strong and weak equivalence of paths

Let us first generalize the storage transformation of path over all the variables of both the source and transformed FSMD. We then present the definition of weak equivalence of paths.

In general, M_0, M_1 may involve different storage variable sets V_0 and V_1, respectively because of code motions. Hence, the equivalence checking should take into account the complete set $V_0 \cup V_1$ of variables. Without loss of generality, the data transformation of any path in M_0 or in M_1 can be extended for the variables in $V_0 \cup V_1$. Specifically, the storage transformation of any path in M_0 or in M_1 is of the form $\langle e_1, \ldots, e_x, e_{x+1}, \ldots, e_y, e_{y+1} \ldots e_z \rangle$, where the sub-tuple $(e_1, \ldots, e_x)$ represents the transformation of the variables in $V_0 - V_1$, the sub-tuple $(e_{x+1}, \ldots, e_y)$ represents the transformation of the variables in $V_0 \cap V_1$ and the sub-tuple $(e_{y+1}, \ldots, e_z)$ represents the transformation of the variables in $V_1 - V_0$. For a path of M_0, $e_i,\ y+1 \leq i \leq z$, would be the symbolic name of the variable v_i (in $V_1 - V_0$). Similarly, for a path of M_1, $e_i,\ 1 \leq i \leq x$, would be the symbolic name of the variable v_i (in $V_0 - V_1$).

Let α be a path in M_0 or in M_1. The tuple s_α restricted to the set V', where $V' \subseteq V_0 \cup V_1$, is its projection over V'. The restriction of s_α of a path α on the variable set V' is denoted as $s_\alpha|_{V'}$. For example, $s_\alpha|_{V_0 - V_1} = (e_1, \ldots, e_x)$ and $s_\alpha|_{V_1 - V_0} = (e_{y+1}, \ldots, e_z)$, where $s_\alpha = \langle e_1, \ldots, e_x, e_{x+1}, \ldots, e_y, e_{y+1} \ldots e_z \rangle$.

As discussed in subsection 3.3.2, the equivalence of paths of two FSMDs are defined in terms of the variables in $V_0 \cap V_1$ in (Karfa, 2007). We term this equivalence as *strong equivalence of paths*. Formal definition of strong equivalence of paths is as follows:

Definition 6 (Strong equivalence of paths) *Two paths β and α are said to be strongly equivalent iff their respective (i) conditions of execution are equivalent, i.e., $R_\beta \equiv R_\alpha$ (ii) output lists are equivalent, i.e., $O_\beta \equiv O_\alpha$, and (iii) storage variable transformations restricted over the variable set $V_0 \cap V_1$ are equivalent, i.e., $s_\beta|_{V_0 \cap V_1} \equiv s_\alpha|_{V_0 \cap V_1}$. The fact that a path β is strongly equivalent to a path α is denoted as $\alpha \simeq_s \beta$.*

Let the respective cardinalities of two output lists O_k and O_l be k and l, where $k \le l$. The list O_k is said to be a prefix of the list O_l, i.e., $O_k = \mathsf{prefix}(O_l)$, iff $O_l = [O_k, O_x]$, where O_x is another output list with length $(l-k)$. These two lists O_k and O_l cannot be equivalent even if we add some more elements in O_k iff $O_k \neq \mathsf{prefix}(O_l)$. Two output lists O_k and O_l are equivalent iff $O_k = \mathsf{prefix}(O_l)$ and $O_l = \mathsf{prefix}(O_k)$.

Let us now define the weak equivalence of paths.

Definition 7 (Weak equivalence of paths) *Let $V' \subseteq V_0 \cup V_1$ be defined as follows. $V' = \{v \mid v$ is transformed in β or α or both and is used before being defined following the end state(s) of β, α or both, respectively $\}$.*
A path β of the FSMD M_0 is said to be weakly equivalent to a path α of FSMD M_1, iff their respective (i) conditions of execution are equivalent, i.e., $R_\beta \equiv R_\alpha$ (ii) output lists are equivalent, i.e., $O_\beta \equiv O_\alpha$, and (iii) storage variable transformations restricted over the variable set V' are equivalent, i.e., $s_\beta|_{V'} \equiv s_\alpha|_{V'}$. The fact that a path β is weakly equivalent to a path α is denoted as $\beta \simeq_w \alpha$.

The paths β and α cannot be weakly equivalent if V' contains a variable v of $V_0 - V_1$ (or $V_1 - V_0$) because such a v has been transformed in only β (or only α), and hence cannot have the same transformation in s_β and in s_α. In other words, $\beta \simeq_w \alpha \Rightarrow s_\beta|_{V'} \equiv s_\alpha|_{V'} \Rightarrow V' \subseteq V_0 \cap V_1$. In contrast, $\beta \simeq_s \alpha \Rightarrow s_\beta|_{V'} \equiv s_\alpha|_{V'}$, where $V' = V_0 \cap V_1$. Consequently, $\beta \simeq_s \alpha$ implies $\beta \simeq_w \alpha$.

A path β is said to be the corresponding path or synonymously, the equivalent path, of α if $\beta \simeq_w \alpha$. Similarly, a computation c_i of FSMD M_0 is said to be the corresponding computation of a computation c_j of FSMD M_1 if $c_i \simeq c_j$.

Based on the definition of weak equivalence of paths, the statement of theorem 1 can be restated as follows.

Theorem 2 *An FSMD M_0 is contained in another FSMD M_1 ($M_0 \sqsubseteq M_1$), if there exists a finite path cover $P_0 = \{p_{0,0}, p_{0,1}, \ldots, p_{0,l}\}$ of M_0 for which there exists a set $P_1 = \{p_{1,0},\ p_{1,1}, \ldots,\ p_{1,l}\}$ of paths of M_1 such that $p_{0,i} \simeq_w p_{1,i}$, $0 \ \leq\ i\ \leq\ l$.*

Proof: $M_0 \sqsubseteq M_1$, if for any computation c_0 of M_0, there exists a computation c_1 of M_1 such that c_0 and c_1 are computationally equivalent [by definition 3].

Now, let there exist a finite cover $P_0 = \{p_{0,0}, p_{0,1}, \cdots, p_{0,l}\}$ of M_0. Corresponding to P_0, let a set $P_1 = \{p_{1,0},\ p_{1,1},\ \cdots,\ p_{1l}\}$ of paths of M_1 exist such that $p_{0,i}\ \simeq_w\ p_{1,i}$, $0 \leq i \leq l$.

Since P_0 covers M_0, any computation c_0 of M_0 can be looked upon as a concatenated path $[p_{0,i_1} p_{0,i_2} \cdots p_{0,i_n}]$ from P_0 starting from the reset state $q_{0,0}$ and ending again at this reset state of M_0. From the above hypothesis, it follows that there exists a sequence Π_1 of paths $[p_{1,j_1} p_{1,j_2} \cdots p_{1,j_n}]$ of P_1 where $p_{0,i_k} \simeq_w p_{1,j_k}$, $1 \leq k \leq n$. So, in order that Π_1 represents a computation of M_1, it is required to prove that Π_1 is a concatenated path of M_1 from its reset state $q_{1,0}$ back to itself representing a computation c_1 such that $c_0 \simeq c_1$. The following definition is in order.

Definition 8 (Corresponding states) *Let $M_0 = \langle Q_0, q_{0,0}, I, V_0, O, f_0, h_0\rangle$ and $M_1 = \langle Q_1, q_{1,0}, I, V_1, O, f_1, h_1\rangle$ be the two FSMDs. (i) The respective reset states $q_{0,0}$ and $q_{1,0}$ are corresponding states. (ii) If $q_{0,i} \in Q_0$ and $q_{1,j} \in Q_1$ are corresponding states and there exist $q_{0,k} \in Q_0$ and $q_{1,l} \in Q_1$ such that, for some path β from $q_{0,i}$ to $q_{0,k}$ in M_0, there exists a path α from $q_{1,j}$ to $q_{1,l}$ in M_1 such that $\beta \simeq_w \alpha$, then $q_{0,k}$ and $q_{1,l}$ are corresponding states.*

Now, let $p_{0,i_1} : [q_{0,0} \Rightarrow q_{0,f_1}]$. Since $p_{1,j_1} \simeq_w p_{0,i_1}$, from the above definition of corresponding states, p_{1,j_1} must be of the form $[q_{1,0} \Rightarrow q_{1,f_1}]$, where $\langle q_{0,0},\ q_{1,0}\rangle$ and $\langle q_{0,f_1},\ q_{1,f_1}\rangle$ are corresponding states. Again, let $p_{0,i_2} : [q_{0,f_1} \Rightarrow q_{0,f_2}]$. By the above argument, $\langle q_{0,f_2},\ q_{1,f_2}\rangle$ are also corresponding states. Since p_{0,i_1} is weak equivalent to p_{1,j_1}, there may exist some variable which is transformed in p_{0,i_1} but not in p_{1,j_1}. The final value of such variables in p_{0,i_1}, however, does not effect any variable or any output of the path p_{0,i_2} [by definition 7]. Similarly, any variable which is transformed in p_{1,j_1} and not in p_{0,i_1} does not effect any variable or any output of the path p_{1,j_2}. Hence, the concatenated paths $p_{0,i_1}p_{0,i_2}$ and $p_{1,j_1}p_{1,j_2}$ should also be weak equivalent, i.e., $p_{0,i_1}p_{0,i_2} \simeq_w p_{1,j_1}p_{1,j_2}$. Thus, by repeated application of the above two arguments, it follows that if $p_{0,i_1} : [q_{0,0} \Rightarrow q_{0,f_1}]$, $p_{0,i_2} : [q_{0,f_1} \Rightarrow q_{0,f_2}], \cdots, p_{0,i_n} : [q_{0,f_{n-1}} \Rightarrow q_{0,f_n} = q_{0,0}]$, then $p_{1,i_1} : [q_{1,0} \Rightarrow q_{1,f_1}]$, $p_{1,i_2} : [q_{1,f_1} \Rightarrow q_{1,f_2}], \cdots, p_{1,i_n} : [q_{1,f_{n-1}} \Rightarrow q_{1,f_n} = q_{1,0}]$, where $\langle q_{0,f_m}, q_{1,f_m} \rangle$, $1 \le m \le n$, are pairs of corresponding states and $p_{0,i_1} \dots p_{0,i_n} \simeq_w p_{1,j_1} \dots p_{1,j_n}$. Hence, Π_1 is a concatenated path representing a computation c_1 of M_1, where $c_1 \simeq c_0$. □

Theorem 2 permits us to adopt the concept of selective path extension to verify the non-uniform code motions. Instead of extending a path blindly when an equivalent path cannot be found or setting the equivalence of paths blindly by ignoring the transformations of the variables which are not in $V_0 \cap V_1$, the path extension will now be guided by some data-flow properties indicating the “defined-used” sequence. In the rest of the chapter, unless mentioned otherwise, the notation $p_1 \simeq p_2$ implies the $p_1 \simeq_w p_2$.

3.4.4 Formulation of the path extension procedure

Following the discussions in the above paragraphs, the path extension procedure is formulated as follows. The equivalence checker first tries to find a strong equivalent path α (in M_1) for β (of M_0). If the equivalence checker fails to find strong equivalence of β, then it tries to find a weak equivalent path of β in M_1. To do so, it first searches for a path α in M_1 which starts from the corresponding state of the start state of β and has the same condition of execution and output list as that of β. As the FSMDs are inherently deterministic, we can have at most one such path α which has the same condition of execution as that of β. After obtaining such an α, the equivalence checker finds the set V' of variables whose transformations in s_β and in s_α are not the same. The transformations of all other variables are the same. Now, if it is the case that the transformations of the variables of V' in β and α have no further use, then α is the weak equivalent path of β. So, the equivalence checker next finds whether the variables in V' are *used before being defined* at $\mathsf{endPtNd}(\beta)$ (i.e., end state of β) and $\mathsf{endPtNd}(\alpha)$. If the property *used before being defined* is false for all the variables in V' at both endPtNd(β) and endPtNd(α), the equivalence checker establishes the weak equivalence of β and α. It is not required to extend β in this case. Conversely, if the property *'used before being defined'* is true for any of the variables in V' at either endPtNd(β) or endPtNd(α), the equivalence checker extends β.

We have discussed in subsection 3.4.3 that the variable set V_0 and V_1 may not be equal due to application of code motion techniques. Let a variable v be present in V_0 of FSMD M_0 but not in V_1 of M_1. Since, v is not in V_1, it is surely not used anywhere in M_1. Therefore, the property v *is used before being defined* is false in all states of FSMD M_1. Continuing the discussion of earlier paragraph, if V' contains v, it is sufficient to check the property v *is used before being defined* only at endPtNd(β). Similarly, if V' contains a variable v', where v' is in V_1 of M_1 but not in V_0 of M_0, it is sufficient to check the property v' *is used before being defined* only at endPtNd(α).

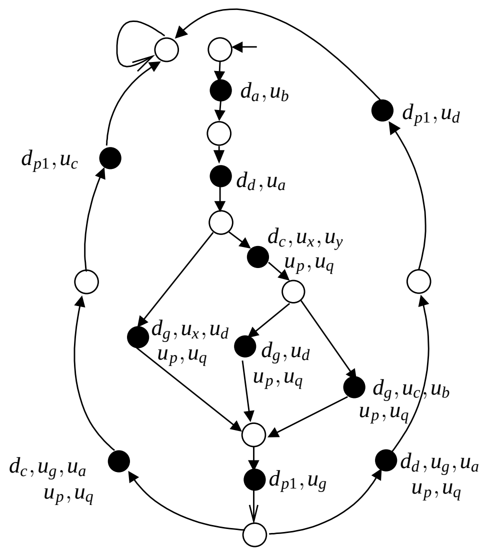


Figure 3.5: Kripke structure obtained from FSMD M_0 of figure 3.1(a)

3.4.5 Encoding and model checking the data-flow properties

In the previous section, we found that the data-flow property *"used before being defined"* should guide the path extension procedure. The question remains how we can determine whether this property holds in a state q in an FSMD. For this purpose, we resort to model checking of the property involving two propositions, d_v and u_v, for each variable v in $V_0 \cup V_1$, where d_v and u_v represent *defined(v)* and *used(v)*, respectively. The required property *“v is used before being defined"* is then encoded as the CTL formula E$[(\neg d_v)$ U $u_v]$. Similar encoding scheme is presented in (Fehnker et al., 2007, 2009) for static analysis of C/C++ programs. The method to convert an FSMD into an equivalent Kripke structure (Clarke et al., 2002) - a step needed for applying model checking - is given as algorithm 1. It is done by some syntactic transformations (whose logical validity is obvious). There is a state in the Kripke structure for each state of the FSMD. There are two modifications needed for depicting an FSMD as a Kripke structure. (i) For any path $q_i \rightarrow q_f$, we need to examine whether the CTL formula E$[(\neg d_v)$ U $u_v]$ hold in q_f or not. The (infinite) paths considered in this process should not extend through the reset state. Hence a dummy state is added in the Kripke structure for the reset state with self loop (so that the paths still remain infinite). All the transitions that terminate in the reset state in the FSMD will be terminated in the dummy state corresponding to the reset state (but not in the reset state) in the Kripke structure. (ii) Since, Kripke structure does not support labels in the transitions, we add a dummy state in the Kripke structure for each transition of the FSMD. For example, figure 3.5 represents the Kripke structure of the FSMD in figure 3.1(a). The dummy states corresponding to the transitions are denoted as black circles in the model. The assignments and the condition expressions are all abstracted out using the propositions d_v and u_v. The proposition d_v will be true in a dummy state if the variable v is defined by some operation in the corresponding transition in the FSMD. Similarly, u_v will be true in a dummy state if the variable v is used in the condition of the transition or in some operation in the transition corresponding to the dummy state in the FSMD. For example, the propositions $d_g,\ u_x,\ u_d$ and u_p, u_q are true in the dummy state in figure 3.5 corresponding to the transition $q_{0,2} \rightarrow q_{0,4}$ in figure 3.1(a) as the variable g is defined and the variables x and d are used in the operation $g \Leftarrow x+d$ in this transition and the variables p and q are used in the condition of the transition is $p > q$. By convention, if any proposition is not present in any state of the Kripke structure, then the negation of the proposition is true in that state. The above property can now be easily verified using any CTL model checker such as NuSMV (Cimatti et al., 2000).

3.4.6 The equivalence checking method

The equivalence checking method is given as algorithm 2. The method constructs an initial path cover P_0 of M_0 by putting cutpoints as stated in subsection 3.3.1. It may

Algorithm 1 FSMDtoKripkeStr(M)

```
/* Input: an FSMD $M$ */
/* Output: a Kripke structure $K$ */
1:  for each state $q_i$ of FSMD $M$ do
2:      add a state $q'_i$ in the Kripke structure $K$; For each variable $v$ in $M$, associate $\neg d_v$ and $\neg u_v$ with the state $q'_i$ in $K$;
3:      if $q_i$ is the reset state, add $q'_{end}$ in the Kripke structure $K$; add a self loop in $q'_{end}$. For each variable $v$ in $V$, associate $\neg d_v$ and $\neg u_v$ with the state $q'_{end}$ in $K$;
4:  end for
5:  for each transition $q_i \xrightarrow[c]{} q_j$ in $M$ do
6:      if $q_j$ is the reset state then
7:          add a state $q'_{icj}$ and two transitions $q'_i \rightarrow q'_{icj}$ and $q'_{icj} \rightarrow q'_{end}$ in $K$;
8:      else
9:          add a state $q'_{icj}$ and two transitions $q'_i \rightarrow q'_{icj}$ and $q'_{icj} \rightarrow q'_j$ in $K$;
10:     end if
11:     Let the condition of the transition $g(v_1, \ldots, v_k)$. Associate $u_{v_i}$, $1 \leq i \leq k$, with the state $q'_{icj}$ in $K$;
12:     for each operation $a \Leftarrow f(v_1, \ldots, v_n)$ associated with the transition $q_i \xrightarrow[c]{} q_j$ do
13:         associate $d_a$ and $u_{v_i}$, $1 \leq i \leq n$, with the state $q'_{icj}$ in $K$;
14:     end for
15:     for each variable $v$ in $M$ do
16:         if $d_v$ is not true in the state $q'_{icj}$ then
17:             associate $\neg d_v$ with the state $q'_{icj}$ in $K$;
18:         end if
19:         if $u_v$ is not true in the state $q'_{icj}$ then
20:             associate $\neg u_v$ with the state $q'_{icj}$ in $K$;
21:         end if
22:     end for
23: end for
```

be noted that initially P_0 contains all the paths from one cutpoint to another without having any intermediate cutpoint. The path cover P_0 is gradually updated in each iteration of the algorithm. The method considers each path β in P_0 from a cutpoint, $q_{0,i}$ say, (whose corresponding state in M_1 is already found) one by one and invokes the function *findEquivalentPath* (given as algorithm 3) with β as an input parameter to find its equivalent path in M_1. The function *findEquivalentPath* checks all the paths in M_1 starting from the state $q_{1,j}$ (which is the corresponding state of $q_{0,i}$) and returns a path α and a Boolean flag EXTEND. It returns one of the following combinations of values of α and EXTEND:

(*i*) the equivalent path of β as α when α is found to be the equivalent path of β,

(*ii*) $\alpha = \mathit{empty}$ and EXTEND = 0 when equivalent of β is not found and cannot be found in M_1 even by extending β, and

(*iii*) $\alpha = \mathit{empty}$ and EXTEND = 1 when the equivalent of β is not found but may be found by extending β.

Based on these returned values, algorithm 2 works as follows:

(*i*) α is non-empty: it puts $\langle \mathsf{endPtNd}(\beta), \mathsf{endPtNd}(\alpha) \rangle$ in the set of corresponding state pairs.

(*ii*) α is empty and EXTEND = 0: the method reports possible non-equivalence of the FSMDs.

(*iii*) α is empty and EXTEND = 1: the method extends β $(= q_{0,i} \Rightarrow q_{0,f})$ in M_0 by concatenating it with all the paths from $q_{0,f}$ to the next cutpoints and puts the extended paths in P_0 in place of β. The method, however, does not allow a path to be extended so that the reset state or any cutpoint occurs more than once as an intermediate node in the extended path. In addition, if the method finds that there is no path of M_0 in P_0 with $\mathsf{endPtNd} = q_{0,f}$, then it removes those paths β' whose $startPtNd(\beta')$ is $q_{0,f}$ from P_0.

The function *findEquivalentPath* (algorithm 3) works as follows.

Algorithm 2 *equivalenceChecker (M_0, M_1)*

```
Require: Two FSMDs $M_0$ and $M_1$
Ensure: $P_0$: a path cover of $M_0$, $E$: ordered pairs $\langle \beta, \alpha \rangle$ of paths of $M_0$ and $M_1$, respectively,
such that $\beta \in P_0$ and $\beta \simeq_w \alpha$.
 1: Let $\zeta$, the set of corresponding state pairs, be $\{\langle q_{0,0}, q_{1,0} \rangle\}$;
 2: Insert cutpoints in $M_0$ using the rules stated in this section. Let $P_0$ be the set of all paths
    of $M_0$ from a cutpoint to a cutpoint having no intermediary cutpoint. Let $E$ be empty;
 3: for each member $\langle q_{0,i}, q_{1,j} \rangle$ of $\zeta$ do
 4:   for each path $\beta \in P_0$ emanating from $q_{0,i}$ do
 5:     $\langle \alpha, \mathtt{EXTEND} \rangle$ = findEquivalentPath($\beta$, $q_{1,j}$, $M_0$, $M_1$);
              /* $\alpha \neq$ empty:: $\alpha$ is the equivalent path of $\beta$. $\alpha =$ empty and EXTEND = 0::
                no equivalent path – nor obtainable by extension of $\beta$. $\alpha =$ empty
                and EXTEND = 1: no equivalent path – but extension of $\beta$ to be tried. */
 6:     if $\alpha$ is not empty then
 7:       $\zeta \leftarrow \zeta \bigcup \{\langle \mathsf{endPtNd}(\beta), \mathsf{endPtNd}(\alpha) \rangle\}$; $E \leftarrow E \bigcup \{\langle \beta, \alpha \rangle\}$;
 8:     else if $\alpha$ is empty then
 9:       if EXTEND is 0 then
10:         Report "equivalent path of $\beta$ may not be present in $M_1$" and exit (failure);
11:       else
12:         Extend $\beta$ $(= \langle q_{0,i} \Rightarrow q_{0,f} \rangle)$ in $M_0$ by concatenating it with all the paths from
            $q_{0,f}$ to the next cutpoints. Let $B_m$ be the set of all such concatenated paths of
            $\beta$;
13:         if any of the paths in $B_m$ contains the reset state or any cutpoint more than once
            as an intermediate node then
14:           Report "$\beta$ may not have any equivalent in $M_1$ and cannot be extended" and
              exit (failure);
15:         else
16:           $P_0 \leftarrow P_0 - \{\beta\}$; $P_0 \leftarrow P_0 \cup B_m$;
17:           For the cutpoint $q_{0,f}$, if there is no path of $M_0$ in $P_0$ with $\mathsf{endPtNd} = q_{0,f}$,
              then $P_0 \leftarrow P_0 - \{\beta' \mid startPtNd(\beta') = q_{0,f}\}$;
18:         end if   // path in $B_m$ contains reset state ...
19:       end if   // EXTEND is 0
20:     end if   // $\alpha$ is not empty
21:   end for   // each path $\beta$ emanating from $q_{0,i}$
22: end for   // each member $\langle q_{0,i}, q_{1,j} \rangle$ of $\zeta$
23: Return $P_0$ as a path cover of $M_0$ and $E$ as a set of ordered pairs of equivalent paths of $M_0$
    (from $P_0$) and $M_1$ and exit (success);
```

(i) A path α is found in M_1 as the strong equivalent of β: the function returns α – the non-emptiness of α indicates that an equivalent path is found.

(ii) The condition of execution of one path matches with that of β but the output lists are not equivalent: the output lists cannot be shown to be equivalent even if β is extended; hence the function returns $\alpha =$ *empty* and EXTEND = 0 which indicates that β has no equivalent path in M_1 and the equivalent path cannot be found even by extending β.

(iii) The condition of execution of one path matches with that of β but one output list is a prefix of the other[3]: equivalence of output lists may be shown by extending β; hence, the function returns $\alpha =$ *empty* and EXTEND = 1.

(iv) a path α_k is found in M_1 which has the same condition of execution and output list as that of β having, however, some variables which are not transformed identically in the paths β and α_k: The function then finds the set V' of variables of β and α_k as defined earlier. The method then calls the function *dataFlowPropertyCheck* (algorithm 4) for both β and α_k to check some data-flow properties (described earlier). The function *dataFlowPropertyCheck*, in turn, obtains the respective Kripke structures from M_0 and M_1 by algorithm 1 and then constructs the appropriate CTL formulas capturing the required property for each variable in V'; it invokes the model checking tool *NuSMV* to check the properties. If for any of the paths, the function *dataFlowPropertyCheck* returns EXTEND, then the function *findEquivalentPath* returns $\alpha =$ *empty* and EXTEND = 1 to the equivalence checker. If, for both the paths, the function *dataFlowPropertyCheck* returns EQUIV (by ensuring that for all the variables in V', the property 'used before being defined' is false), then the function *findEquivalentPath* returns α_k as the weak equivalent path of β to the equivalence checker.

(v) The condition of execution of none of the paths starting from $q_{1,j}$ is found to be the same as that of β: the equivalent path may be obtained by extending β; hence, the function returns $\alpha =$ *empty* and EXTEND = 1.

It may be noted from step 1 of the function $findEquivalentPath$ that the function considers several paths starting from the state $q_{1,j}$ in M_1 to find the equivalence of β

[3] It may be noted that we extend β even if the output list of β strictly covers that of α. The idea is that for each extended path of β, our method eventually finds the extended paths of α with the same output list as that of the extended paths of β.

Algorithm 3 *findEquivalentPath* (β, $q_{1,j}$, M_0, M_1)

/* **Ensure:** a path α and a Boolean flag EXTEND –

α is non-empty – when α is found to be the equivalent path of β;

α is empty and EXTEND = 0 – equivalent of β is not found in M_1 and cannot be found in M_1 by extending β;

α is empty and EXTEND = 1 – equivalent of β is not found in M_1 but may be found by extending β. */

```
1:  for each path $\alpha_k$ which starts from $q_{1,j}$ in $M_1$ do
2:      if $\alpha_k$ is strong equivalent to $\beta$ then
3:          return $\langle \alpha_k, 0 \rangle$;   /* $R_\beta \simeq R_{\alpha_k}$ and $r_\beta \simeq r_{\alpha_k}$. return value of EXTEND is redundant here */
4:      else if $R_\beta \simeq R_{\alpha_k}$ and $O_\beta \simeq O_{\alpha_k}$ then
5:          Let $V'$ be the set of variables whose transformations are not same in $s_\beta$ and $s_{\alpha_k}$;
6:          $x$ = dataFlowPropertyCheck ($\beta$, $V'$, $M_0$);
7:          $y$ = dataFlowPropertyCheck ($\alpha_k$, $V'$, $M_1$);
8:          If $x$ is EXTEND or $y$ is EXTEND: return $\langle \phi, 1 \rangle$;   /* path needs to be extended */
9:          If $x$ is EQUIV and $y$ is EQUIV: return $\langle \alpha_k, 0 \rangle$;   /* $\beta$ are $\alpha_k$ weak equivalent. the return value of EXTEND is redundant here */
10:     else if $R_\beta \simeq R_{\alpha_k}$ then
11:         if $O_\beta = \mathsf{prefix}(O_{\alpha_k})$ or $O_{\alpha_k} = \mathsf{prefix}(O_\beta)$ then
12:             return $\langle \phi, 1 \rangle$;   /* output lists are matched partially; so, equivalent of $\beta$ may found by extending it */
13:         else
14:             return $\langle \phi, 0 \rangle$;   /* output lists are not equivalent; hence equivalent path can not be found even by extend */
15:         end if
16:     end if
17: end for
18: return $\langle \phi, 1 \rangle$;   /* condition mismatches will all the paths from $q_{1,j}$; equivalent path may be found by extending $\beta$ */
```

in M_1. The function considers those paths in the following manner. It first considers all the transitions from $q_{1,j}$. Since all theses transitions have distinct conditions of execution, the condition of execution of only one of them matches (partially) with that of β. If the condition of execution of a transition, $q_{1,j} \rightarrow q_{1,k}$ say, starting from $q_{1,j}$ matches partially with that of β, then the function concatenates the subsequent transitions from $q_{1,k}$ with $q_{1,j} \rightarrow q_{1,k}$ one by one and checks for equivalence. This process continues till any repetition of nodes occurs or the condition of execution of path matches with that of β.

Algorithm 4 *dataFlowPropertyCheck (p, V′, M)*

/* This function checks whether the variables in V' have been used before being defined in the rest of the behaviour in the FSMD M starting from the end state of the path p. It returns EXTEND if any variable of V' is used before being defined; otherwise returns EQUIV */

```
1:  Obtain the Kripke structure K from $M$ using algorithm 1;
2:  for each variable $v \in V'$ do
3:      Let $q_f$ be the end state of the path $p$;
4:      if v is in $V$ then
5:          /* $V$ is the set of storage variables of $M$; if $v$ is not in $V$, the formula $\mathrm{E}[(\neg d_v)\ \mathrm{U}\ u_v]$
            is false by default at $q_f$ */
6:          Check satisfiability of the formula $\mathrm{E}[(\neg d_v)\ \mathrm{U}\ u_v]$ in the state corresponding to $q_f$ in
            Kripke structure K using NuSMV;
7:          if the formula is true in $q_f$ then
8:              return EXTEND;   /* $v$ is used before being defined in the behaviour starting from
                $q_e$ */
9:          end if
10:     end if
11: end for
12: return EQUIV;   /* the formula used before being defined is false all the variables in $V'$ at
    $q_f$*/
```

3.4.7 Illustration of working of the equivalence checking method

Let us now discuss the working of our equivalence checking method with the help of the example given in figures 3.1(a) and 3.1(b). The method first sets the cutpoints in

M_0 and finds the initial path cover $P_0 = \{\beta_1,\ \beta_2,\ \beta_3,\ \beta_4,\ \beta_5,\ \beta_6,\ \beta_7,\ \beta_8\}$. The initial value of $\zeta = \{\langle q_{0,0}, q_{1,0}\rangle\}$. The iterations of the equivalence checking method are as follows:

1. The method first considers the paths from the state $q_{0,0}$. Let $\beta = \beta_1$. It finds a path $\alpha_k = q_{1,0} \rightarrow q_{1,1}$ in M_1 such that both paths have the same condition of execution (which is “true" in this case) and no output but the variable d is transformed in β_1 but not in α_k. The method next finds the property $\mathrm{E}[(\neg d_d)\ \mathrm{U}\ u_d]$ is true at the state $q_{0,1}$. So, β_1 needs path extension. The extended paths of β_1 are $\beta_1\beta_2$ and $\beta_1\beta_3$. Also, the method removes β_2 and β_3 as there is no path in P_0 that terminates in $q_{0,2}$, i.e., the *startPtNd* of β_2 and β_3. Now, $P_0 = \{\beta_1\beta_2,\ \beta_1\beta_3,\ \beta_4,\ \beta_5,\ \beta_6,\ \beta_7,\ \beta_8\}$.

2. Let $\beta = \beta_1\beta_2$. The method finds $\alpha_1 = q_{1,0} \rightarrow q_{1,1} \rightarrow q_{1,2} \rightarrow q_{1,5}$ in M_1 as the strong equivalent of β. The method sets the end states of β and α_1, i.e., $\langle q_{0,4}, q_{1,5}\rangle$, as a corresponding state pair and puts it in ζ.

3. Let $\beta = \beta_1\beta_3$. The method fails to find any path in M_1 with same condition of execution. So, β needs path extension. The method also removes β_4 and β_5 from P_0 as there is no path in P_0 which terminates in $q_{1,3}$. Now, $P_0 = \{\beta_1\beta_2,\ \beta_1\beta_3\beta_4,\ \beta_1\beta_3\beta_5,\ \beta_6,\ \beta_7, \beta_8\}$.

4. Let $\beta = \beta_1\beta_3\beta_5$. The method finds $\alpha_{21} = q_{1,0} \rightarrow q_{1,1} \rightarrow q_{1,4} \rightarrow q_{1,5}$ with the same condition of execution and output list (empty) but the variable d is transformed in $\beta_1\beta_3\beta_5$ but not in α_{21}. Next, the method finds that the property $\mathrm{E}[(\neg d_d)\ \mathrm{U}\ u_d]$ is false for both $q_{0,4}$ (= $\mathsf{endPtNd}(\beta)$) and $q_{1,5}$ (= $\mathsf{endPtNd}(\alpha_{21})$). So, α_{21} is weak equivalent to the path β.

5. Similarly, it can be shown that the path $\alpha_{22} = q_{1,0} \rightarrow q_{1,1} \rightarrow q_{1,3} \rightarrow q_{1,5}$ is weak equivalent to the path $\beta_1\beta_3\beta_4$ for a similar reason for the variable c. At this stage, the equivalent of all the paths from the state $q_{0,0}$ found and hence the corresponding pair $\{\langle q_{0,0}, q_{1,0}\rangle\} \in \zeta$ is marked.

6. The method now considers paths from state $q_{0,4}$. It finds $\alpha_3 = q_{1,5} \rightarrow q_{1,6}$ as the equivalent path of β_6. The method sets the end states of β_6 and α_3, i.e., $\{\langle q_{0,4}, q_{1,6}\rangle\}$ in ζ. There is no other path the state $q_{0,0}$ and hence the corresponding pair $\langle q_{0,4}, q_{1,5}\rangle \in \zeta$ is marked.

7 and 8. The method now considers paths from state $q_{0,5}$. In the next two iterations, the method finds $\alpha_4 = q_{1,6} \xrightarrow[\neg p=q]{} q_{1,8} \rightarrow q_{1,0}$ and $\alpha_5 = q_{1,6} \xrightarrow[p=q]{} q_{1,7} \rightarrow q_{1,0}$ as the respective equivalent paths of β_8 and β_7. The method next marks the corresponding pair $\langle q_{0,5}, q_{1,6} \rangle$. Now, all corresponding state pairs of ζ have been marked and each member of P_0 have an equivalent path in M_1. Hence, the method establishes the equivalence between these two FSMDs.

Let us now consider the equivalence checking between the FSMD in figure 3.1(c) with no operation in the transition $q_{0,5} \xrightarrow[\neg p=q]{} q_{0,7}$ and the FSMD in figure 3.1(d). For this case, the first three iterations of the algorithm would be exactly the same as the first three iterations of the algorithm for equivalence checking between FSMDs in figures 3.1(a) and in figure 3.1(b) (as given above). In step 4, the path $\beta = \beta_1\beta_3\beta_5$, however, needs to be extended in this case as the method finds the property $\mathrm{E}[(\neg d_d)\ \mathrm{U}\ u_d]$ to be true at end point of $\alpha_{21} = q_{1,0} \rightarrow q_{1,1} \rightarrow q_{1,4} \rightarrow q_{1,5}$. The extended path of β is $\beta_1\beta_3\beta_5\beta_6$. The method again finds the path $\alpha_{211} = q_{1,0} \rightarrow q_{1,1} \rightarrow q_{1,4} \rightarrow q_{1,5} \rightarrow q_{1,6}$ with the same condition of execution and output list but the variable d is transformed in $\beta_1\beta_3\beta_5\beta_6$ but not in α_{211}. The method next finds that the property $\mathrm{E}[(\neg d_d)\ \mathrm{U}\ u_d]$ is again true at $q_{1,7}$. So, this path is extended and the extended paths are $\beta_1\beta_3\beta_5\beta_6\beta_7$ and $\beta_1\beta_3\beta_5\beta_6\beta_8$. For the path $\beta_1\beta_3\beta_5\beta_6\beta_8$, the method cannot find any equivalent path in M_1 starting from $q_{1,0}$ because of the mismatch of the output. Hence, the method reports a possible non-equivalence between these two FSMDs.

3.4.8 Justification of the initial cutpoints

As discussed in subsection 3.3.1, we choose the reset state, and all the convergence and divergence states of an FSMD as cutpoints. Choice of cutpoints, however, is not unique and it is not guaranteed that a path cover of one FSMD obtained from any choice of cutpoints in itself will have the corresponding set of equivalent paths for the other FSMD. It is, therefore, necessary to search for a ‘suitable’ choice of cutpoints such that the path cover obtained by them have a set of equivalent paths in the other FSMD. Specifically, a suitable choice of cutpoints finally establishes the equivalence of two FSMDs by theorem 2. We start from a ‘good’ set of cutpoints and then arrive at the ‘suitable’ set of cutpoints. A set of cutpoints is a ‘good’ starting point if we can reach the ‘suitable’ one from such a set efficiently with minimum number of iterations in most of the cases. In the following, we discuss why our choice of cutpoints is a 'good starting point' for verification of code motion transformations.

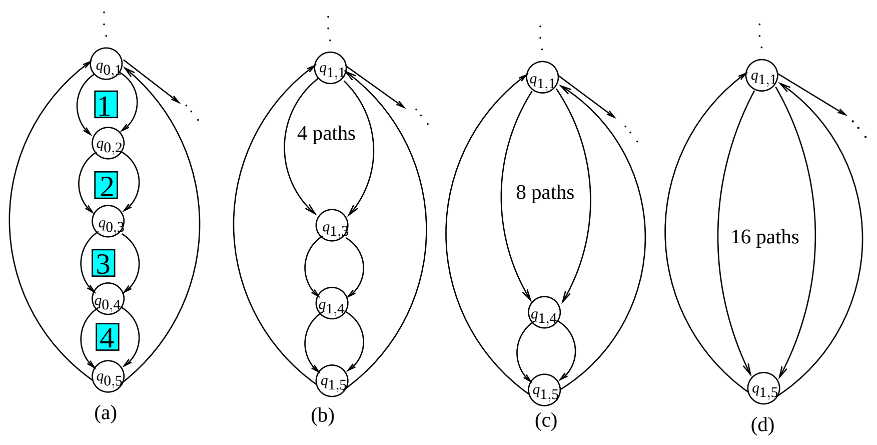


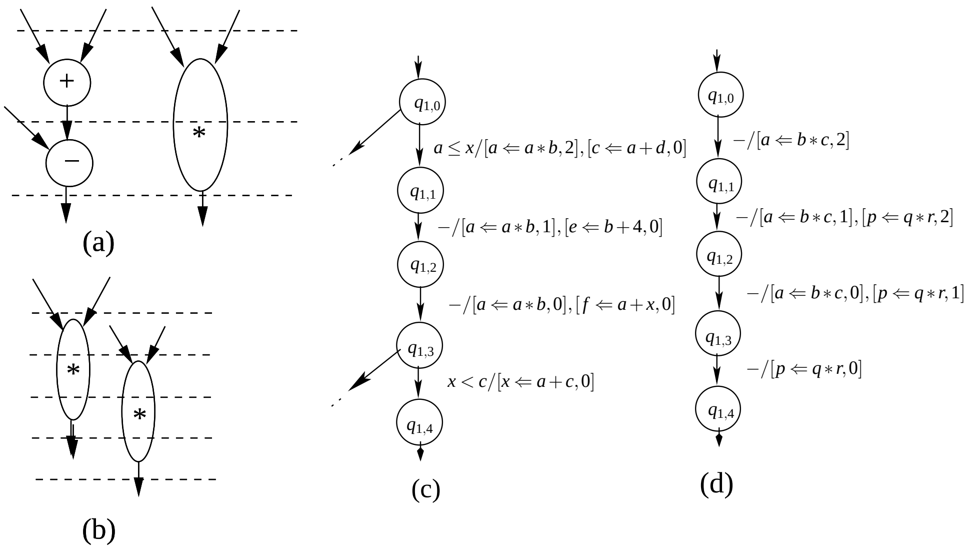
Figure 3.6: An example of cutpoint selection

It might be noted that we have chosen some redundant points as the initial cutpoints. The situation is clarified by figure 3.6(a). There is only one loop in the figure. We can cut this loop by choosing only state $q_{0,1}$ as a cutpoint. (In other words, in general, we can choose the reset state and all the loop starting states as cutpoints.) It gives us the minimum number of cutpoints. Instead, we choose all the five states as cutpoints. If we choose $q_{0,1}$ as the only cutpoint, there would be 32 paths in the initial path cover. Specifically, each path comprises five transitions. For each transition, we have two choices; either taking the left transition or taking the right transition. In each iteration, algorithm 2 finds an equivalent of each of the 32 paths - the algorithm takes 32 iterations to show the equivalence between the FSMDs. In contrast, if we choose all the states as cutpoints (since they represent merging of control flows), then there would be 10 paths in the initial path cover. Consider the code segments in four conditional blocks between $q_{0,1}$ and $q_{0,2}$ (segment 1), between $q_{0,2}$ and $q_{0,3}$ (segment 2), between $q_{0,3}$ and $q_{0,4}$ (segment 3) and between $q_{0,4}$ and $q_{0,5}$ (segment 4). The segment numbers are shaded in the figure. It may happen that the scheduler (*i*) does not move any code beyond the conditional block boundaries, (*ii*) moves code between the first and the second conditional blocks, (*iii*) moves code between the first and the third conditional blocks or (*iv*) moves code between the first and the fourth conditional blocks. For case (i), no path extension is required. Our algorithm 2 finds an equivalent of each of the 10 paths of the path cover in each iterations. So, the algorithm takes 10 iterations to establish the equivalence. Let us now consider the case (ii). Algorithm 2 extends two paths in segment 1 to segments 2 in two iterations and put four extended paths in P_0. It also removes paths in segments 1 and 2 from P_0 So, after extensions P_0 contains 10 paths. In the next 10 iterations, algorithm 2 finds an equivalent of each of the 10 paths of the path cover. So, algorithm 2 takes 12 iterations. In the same way, one can show that for cases (iii) and (iv), the algorithm takes respectively, 18 or 32 iterations. The numbers of paths in the path cover (and hence the actual cutpoints) are shown in figures 3.6(a), 3.6(b), 3.6(c), 3.6(d), respectively for these four cases. Even though the algorithm takes the same number of iterations in the last case (as that in the original case), its performance is better in all other cases. Also, it may be noted that the chance of moving code between the first and the fourth conditional blocks is the least compared to all other cases.

3.5 Multicycle and pipelined execution of operations

Real functional units have different propagation delays based on their designs. A multiplier, for example, is much slower than an adder. Therefore, all the operations do not finish in one control step. Since an operation may no longer execute in one cycle, we can have the following two execution models of operations. 1. *Multicycle* execution of an operation requires two or more control steps (figure 3.7 (a)). This increases the utilization of the faster functional units since two or more operations can be scheduled on them during one operation on the multicycle units. 2. Pipelining the pairs of operands over the cycles of a multicycle execution of an operation p allows concurrent execution of p on these pairs of operands, each pair getting partially computed in one stage of the pipelined unit (figure 3.7 (b)) resulting in a better utilization of the (pipelined) unit. An operation may be scheduled in multicycle or pipelined manner when the code motion techniques are applied in the behaviour. The definition of FSMD model and our verification algorithm should, therefore, accommodate multicycle and pipelined operations. In the following, we discuss the enhancement of FSMD models and the computation of data transformation for this purpose.

Our algorithm can handle multicycle and pipelined execution of the operations with the following modification of the update function in the FSMD definition in section 3.2.1: $h : Q \times 2^{\mathcal{S}} \rightarrow U \times \mathbb{N}$, where $Q, \mathcal{S}, U$ are the same as before and $\mathbb{N}$ represents the set of natural numbers. Specifically, $h(q, true) = (a \Leftarrow a * b, t)$ represents that t more time steps are required to complete the operation '*'; this second



(a)
(b)
$q_{1,0}$
$a \leq x/[a \Leftarrow a*b,2],[c \Leftarrow a+d,0]$
$q_{1,1}$
$-/[a \Leftarrow a*b,1],[e \Leftarrow b+4,0]$
$q_{1,2}$
$-/[a \Leftarrow a*b,0],[f \Leftarrow a+x,0]$
$q_{1,3}$
$x < c/[x \Leftarrow a+c,0]$
$q_{1,4}$
(c)
$q_{1,0}$
$-/[a \Leftarrow b*c,2]$
$q_{1,1}$
$-/[a \Leftarrow b*c,1],[p \Leftarrow q*r,2]$
$q_{1,2}$
$-/[a \Leftarrow b*c,0],[p \Leftarrow q*r,1]$
$q_{1,3}$
$-/[p \Leftarrow q*r,0]$
$q_{1,4}$
(d)

Figure 3.7: (a) A 2-cycle multiplier; (b) A 3-stage pipelined multiplier; (c) A sample path in an FSMD with multicycle operation; (d) A sample path in an FSMD with pipelined operation

member t of the value-tuple of h is called the *span* of that operation. For a single cycle operation, t should be zero. The input behaviour does not specify whether an operation is multicycle or pipelined. Hence, all operations in the input FSMD are represented as single cycle ones. Depending upon the type of the operators, the operations are scheduled in a multicycle or pipelined manner by the scheduler. So, these would be reflected in the scheduled FSMD. In figure 3.7(c), the operation $a \Leftarrow a*b$ is scheduled in a 3 cycle multiplier. In the figure 3.7(d), the operations $a \Leftarrow b*c$ and $p \Leftarrow q*r$ are scheduled using a 3 stage pipelined multiplier. The value produced by a multicycle or a pipelined operation is available only when the span of the operation becomes zero. So, during computation of the condition of execution and the data transformations of a path in a scheduled FSMD, an operation will be considered only when its span value becomes zero. For example, for the path shown in figure 3.7(c), the condition of execution is $a \leq x \wedge x < (a+d)$ and the data transformation is $\langle\langle a*b,\ b,\ a+d,\ d,\ b+4,\ a+x,\ a*b+a+d\rangle, -\rangle$, where the variables are in order $\langle a \prec b \prec c \prec d \prec e \prec f \prec x\rangle$. It may be noted that the data transformation corresponding to the operation $f \Leftarrow a+x$ uses the old value a whereas the operation $x \Leftarrow a+c$ uses the updated value of a.

Since the operations are now scheduled over multiple steps, the execution of an operation should complete within a path. In other words, a cutpoint should not be placed in an FSMD such that execution of an operation spans over multiple paths. The presence of multicycle and(or) pipelined operations, however, do not create any problem in selecting the cutpoints. Let a 3-cycle operation p be conceived in the pre-scheduled FSMD in a transition $q_l \rightarrow q_m$; let, in the scheduled FSMD, p spans over the transition sequence $q_i \rightarrow q_j \rightarrow q_k \rightarrow q_s$, where q_i corresponds to q_l and q_s corresponds to q_m. It is obvious that there cannot be any bifurcation of flow from the intermediary states q_j or q_k in the scheduled behaviour. For the pipelined operation, however, another aspect is to be checked: whether the sequence in which the operand pairs appear at the input stage of the pipeline is in keeping with that in which the destination registers are strobed. This aspect is verified in the RTL generation phase which is discussed in the next chapter.

3.6 Correctness and complexity

3.6.1 Correctness

Let the cutpoints be put in an FSMD M according to the rule given in subsection 3.3.1. Let P be a set of all paths from one cutpoint to another without having any intermediary cutpoint. Hence, by Floyd-Hoare method of program verification (Floyd, 1967; Hoare, 1969; King, 1980), P is a path cover of M. Let us now define two operations on the set P. We will show that P remains a path cover of M after any number of applications of these two operations.

Definition 9 (Concatenation) *Let p be a path in P. Let* `endNdPt(p)` *denote the end state of p. Let P_t be the set of all paths in P that start from* `endNdPt(p)`. *Let P_p be the set of paths which are obtained by concatenating p with the members of P_t. Replace p with P_p in P.*

Lemma 1 *Let P' be a set of paths obtained from a path cover P by an application of concatenation operation to a member of P. The set P' is a path cover.*

Proof: Let c be any computation in M that contains some path p on which the concatenation operation has been applied. Let c be represented as the concatenated paths $\pi_1 = [p_{j_1} p_{j_2} \dots p_{j_n}]$, where $c \simeq \pi_1$ and $p_{j_i} = \langle q_{j_{i-1}} \Rightarrow q_{j_i} \rangle$, $1 \le i \le n$, be the paths of P; let p_{j_k} be the path p. So, in particular, p is of the form $p_{j_k} = \langle q_{j_{k-1}} \Rightarrow q_{j_k} \rangle$ and the next path in π_1 is $p_{j_{k+1}} = \langle q_{j_k} \Rightarrow q_{j_{k+1}} \rangle$. Since p is concatenated with all its successor paths in P, there must exist a path p'_k in P' of the form $\langle q_{j_{k-1}} \Rightarrow q_{j_{k+1}} \rangle$. So, corresponding to π_1, there exists a concatenated path π_2 of the form $\pi_2 = [p_{j_1} p_{j_2} \dots p_{j_{k-1}} p'_k p_{j_{k+2}} \dots p_{j_n}]$, such that, $\pi_1 \simeq \pi_2$. Therefore, $c \simeq \pi_2$. Since, c is chosen arbitrarily, the above observation holds for any computation. Hence, P' is a a path cover. □

Definition 10 (Deletion) *Let there be a cutpoint q in M such that no path in (a path cover) P terminates in q, Remove all the paths starting at q from P.*

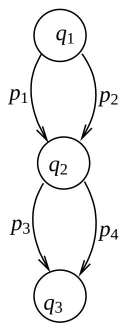


Figure 3.8: Illustration of deletion operation

Let us now analyze how we can reach a situation where there is a cutpoint in which no path of the path cover terminates. Let us consider the FSMD in figure 3.8 for this purpose. The states q_1, q_2 and q_3 of this FSMD are cutpoints in our logic. The path cover of the FSMD is $P = \{p_1,\ p_2,\ p_3,\ p_4\}$. Now, let the *concatenation* operation be applied on path p_1. So, after this operation, the path cover becomes $P' = \{p_1p_3,\ p_1p_4,\ p_2,\ p_3,\ p_4\}$. Now, if the *concatenation* operation be applied on p_2, then the path cover is $P' = \{p_1p_3,\ p_1p_4,\ p_2p_3,\ p_2p_4,\ p_3,\ p_4\}$. Since all the paths that terminate in q_2 are extended, there is no path in P' that terminates on q_2. All computations in this FSMD which follow the path p_3 or p_4 are covered by the first four members and accordingly members p_3 and p_4 need no longer be maintained. So, we can remove them from the path cover and the updated path cover is $P' = \{p_1p_3,\ p_1p_4,\ p_2p_3,\ p_2p_4\}$. Hence we have the following lemma.

Lemma 2 *Let P' be a set of paths obtained from a path cover P by an application of deletion operation to a member of P. The set P' is a path cover.*

Proof: Let there be no path in P that terminates in a state, q say. Let c be any computation which can be represented as the concatenated paths $\pi_1 = [p_{j_1} p_{j_2} \ldots p_{j_n}]$, where $p_{j_i} = \langle q_{j_{i-1}} \Rightarrow q_{j_i} \rangle$, $1 \leq i \leq n$, be the paths of P and $q_{j_0} = q_{j_n} =$ the reset state. Since there is no path in P that terminates in q, none of the q_{j_i}, $0 \leq i \leq n$, is q. Hence, P' remains a path cover after removing all paths starting from q. Therefore, P' remains a path cover after application of *deletion* operation. □

Theorem 3 (Termination) *Algorithm 2 always terminates.*

Proof: In each iteration of the loop (5-21) in algorithm 2, either the equivalent of the path β of M_0 is found in M_1 or the equivalent of β is not found in M_1 and β is extended. (The other case where β is decided to be not worth extending leads to (failure) exit). If a path is considered in one iteration of the algorithm, it will not be considered again in other iterations of the loop because of the restriction of path extension imposed in step 14. Let at any iteration, P_c be the set of paths of M_0 that are already considered by the algorithm. Initially, P_c is empty. The cardinality of P_c increases by one in each iteration (not leading to (failure) exit) of the algorithm. Let P_{all} be the set of all possible paths between cutpoints (with or without intermediary cutpoints) in FSMD M_0. The set P_{all} is finite because of the condition imposed by step 13. Hence, each execution of the loop reduces $\|P_{all} - P_c\|$ by one. Since $\|P_{all} - P_c\|$ is in the well-founded set (Manna, 1974) of non-negative numbers having no infinite decreasing sequence, the loop (5-21) of algorithm 2 cannot execute infinitely long.

Now consider the outer loop (3-22) in algorithm 2. In each iteration of this loop, one corresponding state pair from the set ζ is considered. Corresponding state pairs are added in ζ through *set union* in step 7. Hence, no corresponding state pair is considered twice. Let the the number of states in FSMDs M_0 and M_1 be n_1 and n_2, respectively. Therefore, the upper bound of the number of corresponding state pairs is $n_1 \times n_2$. Let at any iteration, n_ζ be the number of corresponding state pairs that are already considered by the algorithm. Initially, n_ζ is zero. In each iteration of the loop (3-22), n_ζ increases by one. Therefore, each execution of the loop (not leading to (failure) exit) reduces $n_1 \times n_2 - n_\zeta$ by one. Since $n_1 \times n_2 - n_\zeta$ is in the well-founded set (Manna, 1974) of non-negative numbers having no infinite decreasing sequence, the loop (3-22) of algorithm 2 cannot execute the loops infinitely long. Hence, the algorithm also terminates. $\square$

Let C be the set of all the cutpoints in M_0 obtained in step 2 of algorithm 2 by the rule given in section 3.4.6. Let $C' \subseteq C$ be such that every cutpoint in C' has a corresponding state in M_1. We never extend a path beyond a loop. Therefore, all the loops in the FSMD M_0 are cut by at least one cutpoint in C'. Also, algorithm 2 never allows the path β of M_0 to be equivalent of α of M_1 when the conditions of execution and the output lists are not the same in them (ensured by the invoked function $findEquivalentPath$). Step 7 of algorithm 2 ensures that the set P_0 contains paths of the form $\langle q_{0l} \Rightarrow q_{0m} \rangle$, where $q_{0l}, q_{0m} \in C'$. But, the converse is not true, i.e., P_0 may not contain all such paths; moreover, P_0 may contain paths which have intermediary states belonging to C'. So, the final value of P_0 does not satisfy the Floyd-Hoare rules of path cover. We prove by the following theorem that P_0 remains a path cover of M_0 even if it does not satisfy Floyd-Hoare rules of path cover.

Theorem 4 (Soundness) *If algorithm 2 terminates, then $M_0 \sqsubseteq M_1$.*

Proof: From theorem 2, it follows that $M_0 \sqsubseteq M_1$, if the output value of P_0 yielded by algorithm 2 is a path cover of M_0. Recall that E is the set of ordered pairs of equivalent paths of M_0 and M_1. Steps $5-7$ of the algorithm ensure that the output E of the algorithm contains only pairs of equivalent paths of M_0 (belonging to P_0) and M_1; this property of E, therefore, is an invariant. So, we have to show that the assertion, P_0 (the other output of the algorithm) is a path cover of M_0, is an invariant at each step of the algorithm. Specifically, we prove the above assertion for entry (step 3) of the outer loop (3-22). Since, the algorithm 2 is assumed to terminate, the loop (3-22) is finally exited; once proved, the invariant that P_0 is a path cover of M_0 ensures that the exit takes place from step 3 with P_0 as a path cover.

At step 2, P_0 is a path cover by Floyd-Hoare's rule of path cover. As there is no statement other than the loop body (4-21) in the outer loop (3-22), we have only to prove that the assertion is true at the entry point (step 4) of the inner loop (4-21) of the algorithm. Hence, let us consider the assertion $\mathcal{A}(n)$: P_0 is a path cover on the n^{th} entry of the loop (4-21). We prove $\forall n \mathcal{A}(n)$ by induction on n.

Basis (n=1): Step 4 is entered first from step 2 and 3 with value of P_0 which is a path cover by Floyd-Hoare rule.

Induction step: Hypothesis: Let step 4 be entered m^{th} time with a value of $P_0^{(m)}$; $P_0^{(m)}$ is a path cover.

Let $n = m+1$. Let us now analyze how $P_0^{(m+1)}$ is obtained from $P_0^{(m)}$ in the m^{th} iteration of the loop. In particular, the following cases may arise: (i) The equivalent path of β is found: The iteration does not make any change in $P_0^{(m)}$. Therefore, $P_0^{(m+1)} = P_0^{(m)}$ remains a path cover at the $(m+1)^{th}$ entry to the loop. (ii) The equivalent of β is not found: In steps 12-16, the *concatenation* operation is performed on $P_0^{(m)}$ resulting in P_0', say. By lemma 1, P_0' is a path cover of M_0. In step 17, it applies *deletion* operation on P_0' resulting in $P_0^{(m+1)}$. From lemma 2, $P_0^{(m+1)}$ remains a path cover of M_0 subsequently. $\square$

3.6.2 Complexity

The complexity of normalization of a formula F is $O(\|F\|^2)$, where $\|F\|$ denotes the length of the formula, due to multiplication of normalized sums. (For all other operations, it is linear in $\|F\|$). Let n be the number of states in the FSMD and k be the maximum number of parallel edges between any two states. So, the maximum possible state transitions from a state is $k.n$. The number of edges in an FSMD is kn^2 in the worst case and kn in the best case. In the Kripke structure, we have one state for each state of the FSMD and one state and two edges for each edge of FSMD. The formula length is fixed here. The complexity of CTL model checking is $O(|\psi|.(x+y))$, where $|\psi|$ is the length of the CTL formula ψ and x and y are respectively the number of states and the number of edges in the Kripke structure. In our case, the CTL model checking complexity is $O(n^2)$ in the worst case and $O(n)$ in the best case.

Let us find the complexity of the the function $findequivalentPath$. All the transitions emanating from a state have distinct conditions of execution. As discussed in subsection 3.4.6, the condition of at most one of these transitions from q_{1j} matches (may be partially) with R_β. If the condition of transition matches partially, then the function will concatenate the subsequent transitions with this path one by one and check for equivalence. This process will continue till any repetition of nodes occurs or the condition of execution of a path matches. In the worst case, this process iterates n times. The CTL model checker may be invoked to check some data-flow properties only when the condition of execution matches. The condition of execution of at most one path may match. So, the complexity of finding the equivalent path is $O(kn.n.\|F\|^2 + n^2) = O(kn^2\|F\|^2)$, where $\|F\|$ is the maximum of the lengths of R_β, R_α, r_β and r_α. However, the equivalent path can be found in $O(1.\|F\|^2)$ time when there is only one path from q_{1j} which is equivalent to β.

It is required to find the equivalent path for every path in P_0. Initially, this set contains at most $O(n^2)$ number of paths. In the best case, the equivalent path for each member of this set can be found directly and no path extension is required. Also, the number of paths is $O(n)$ (for structured flow-chart). In the worst case, one path may be required to be extended n times. In this case, we have to consider $k.(n-1) + k^2.(n-1).(n-2) + \ldots + k^{(n-1)}.(n-1).(n-2). \ldots .2.1 \simeq k^{(n-1)}.(n-1)^{(n-1)}$ number of paths.

So, the complexity of our algorithm is $O(k^{(n-1)}(n-1)^{(n-1)}.\ kn^2.\ \|F\|^2) = O(k^n n^{(n+1)}.\|F\|^2)$ in the worst case and $O(n.1.\|F\|^2) = O(n\|F\|^2)$ in the best case.

3.7 Experimental results

The verification method described in this paper has been implemented in C and tested on the results produced by a high-level synthesis tool SPARK (Gupta et al., 2003c) that employs a set of code transformations during scheduling. Specifically, SPARK employs common sub-expression elimination (CSE), copy propagation, dead code elimination in its pre-synthesis phase, dynamic renaming of variables and all kinds of code motion techniques during scheduling phase. The FSMDs are extracted from the behaviours at input and the output of the scheduling phase of SPARK. Our equivalence checker has been run for fourteen HLS benchmarks (Panda and Dutt, 1995; Parker et al., 1986; Wakabayashi and Yoshimura, 1989). The characteristics of the benchmarks in terms of the numbers of basic blocks (BB) and branching blocks are

			#operation		*#variable*		*#lines of code*	
Benchmark	*#BB*	*#branch*	*before*	*after*	*before*	*after*	*before*	*after*
GCD	7	5	9	29	5	16	26	96
TLC	17	6	28	58	13	31	79	183
MODN	7	4	12	22	6	17	18	67
PERFECT	6	3	8	13	4	11	17	48
LRU	22	19	49	74	19	61	113	227
DHRC	7	14	131	231	72	209	169	543
BARCODE	28	25	52	74	17	58	151	310
FINDMIN8	14	7	21	24	15	22	35	94
IEEE754	40	28	74	237	16	135	223	558
PRAWN	85	53	227	652	29	433	487	1484
DIFFEQ	4	1	19	20	12	18	23	60
WAKA	6	2	17	21	32	36	31	58
PARKER	14	6	22	33	14	31	39	102
QRS	26	16	139	125	35	112	178	332

Table 3.1: Characteristics of the HLS benchmarks and synthesis results of SPARK

listed in the second and the third columns in table 3.1. The number of three-address operations, variables and lines of code in the input C code of the behaviours and in the C code at the output of the scheduling phase of SPARK are also listed in table 3.1. These figures are taken during our first experiment explained below. It might be noted from the table that the behaviours are transformed significantly by the synthesis tool. The model checking tool NuSMV (Cimatti et al., 2000) has been integrated in our equivalence checker for checking the satisfiability of the data-flow properties.

In our first experiment, we enabled all possible code motions in SPARK. The benchmark behaviours are run on SPARK and the FSMDs are constructed automatically from the initial behaviour and the intermediate results after the scheduling steps of SPARK for all the benchmarks. Table 3.2 lists the verification results in terms of the number of states in the input FSMD (M_0) and that in the output FSMD (M_1), the number of paths in the computed path cover (P_0) of M_0, the number of path extensions required during equivalence checking, the number of times the model checker is invoked, the number of weak equivalent paths detected by our method, the number of iterations of the algorithm and the average overall execution time and time taken for model checking of the algorithm. It may be noted that path extension is required to handle code motions beyond basic blocks. So, it is evident from column

	#states		*verification results*					*time(s)*	
Benchmarks	M_0	M_1	*#paths(P_0)*	*#Extend*	*#modelChk*	*#weakEquiv*	*#iterations*	*overall*	*modelChk*
(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)
GCD	7	6	11	4	5	0	17	0.325	0.294
TLC	13	13	20	0	2	1	20	0.245	0.184
MODN	6	5	9	4	2	0	13	0.541	0.452
PERFECT	9	6	7	1	2	0	8	0.107	0.078
LRU	33	25	39	1	14	4	39	3.570	3.449
DHRC	62	47	31	1	18	1	33	162.646	162.000
BARCODE	32	29	55	2	10	4	64	3.131	3.011
FINDMIN8	8	9	15	45	32	7	32	10.663	10.378
IEEE754	55	42	59	13	42	9	74	345.032	344.238
PRAWN	122	114	154	1	4	2	154	582.068	581.650
DIFFEQ	16	10	3	1	5	2	3	0.269	0.201
WAKA	9	12	5	2	10	2	5	1.788	1.728
PARKER	12	10	13	8	35	5	23	6.562	6.436
QRS	53	24	35	51	66	8	519	194.000	183.800

Table 3.2: Results for several high-level synthesis benchmarks

5 of the table that SPARK has applied code motion techniques in all the cases except one. Column 6 shows how frequent the model checker is invoked during equivalence checking. The model checker decides whether to extend a path or not based on the '*used before being defined*' property for each variable whose transformation is not equal in the corresponding paths. So, the model checker is invoked quite often by the method. Column 7 shows the number of times the model checker has explored weak equivalence of paths. The weak equivalence of paths arises due to a valid non-uniform code motion. It is evident from column 7 of the table that non-uniform transformations have been applied in most of the cases. So, our method successfully verifies the non-uniform code motions also. As discussed in the introduction section, the competitive methods (Karfa, 2007; Kim and Mansouri, 2008) could not handle non-uniform code motions and therefore, they are likely to produce false-negative results in these cases. The number of iterations listed in column 8 suggests that the upper bound of complexity of the algorithm is not hit for the practical scheduling verification cases. The overall execution time includes the FSMD construction time, equivalence checking time, converting the FSMD model to the Kripke structure in NuSMV input format and the property checking time of NuSMV. We have run each example at least six times to get the average overall execution time. Model checking time includes the

time to convert an FSMD to an NuSMV model and the model checking time using NuSMV tool. It may be noted that major portion of the execution time is spent for model checking. The size of the NuSMV model depends on the size of the FSMD and on the number of variables in the behaviour. The NuSMV model for the IEEE754 input behaviour, for example, contains over nine thousand lines of code. Therefore, creating a file with NuSMV model from an FSMD takes large amount of the model checking time. For large FSMDs like for DHRC, IEEE754, PRAWN, etc., the model checking time, therefore, is comparatively high. However, as is evident from column 9 of the table, the time required for verification is not very high and is well within in acceptable range.

	common results		*time (sec)*		*modelCheck*
Benchmarks	*#iteration*	*#extend*	*(Karfa, 2007)*	*Our method*	*Our method*
DIFFEQ	3	0	0.013	0.026	0
GCD	11	0	0.024	0.040	0
TLC	20	0	0.035	0.052	0
MODN	11	3	0.037	0.530	8
PERFECT	7	0	0.015	0.027	0
LRU	39	3	0.730	3.355	14
DHRC	27	3	0.074	52.304	4
BARCODE	55	2	0.098	1.392	4
IEEE754	73	6	0.178	64.790	12
PRAWN	158	4	0.546	2962.000	44

Table 3.3: Results for verification of uniform code motions

	time(sec)		
Benchmarks	*our method*	*method in (Kundu et al., 2010)*	*method in (Kim and Mansouri, 2008)*
DIFFEQ	0.026	1.680	-
TLC	0.052	-	77.457
GCD	0.040	-	7.673
FINDMIN8	4.110	14.860	66.056
WAKA	1.230	2.610	-
PARKER	2.042	5.230	-

Table 3.4: Comparison time with competitive methods for uniform code motions

In our second experiment, we allow the SPARK scheduler to perform only uniform code transformations. The SPARK tool has such provision for the users. The purposes of this experiment are (i) identifying the timing overhead of our method over the method reported in (Karfa, 2007) and (ii) comparing the performance of our method with the competitive methods reported in (Kundu et al., 2010) and (Kim and Mansouri, 2008). It may be noted that our method is an enhancement of the method reported in (Karfa, 2007) which also works for uniform code motions. In the case of uniform code motion, each iteration of our method finds either a strong equivalent path of the path under consideration or extends the path. In the later case, our method performs the property checking for all the variables, whose transformations mismatch to find a possible weak equivalent path before extending the path. However, the model checking is redundant here because we cannot have a weak equivalent path for uniform code motions. So, our method performs some redundant property checking in this case. As a result, it takes more time compared to the time taken by the method reported in (Karfa, 2007). The results tabulated in table 3.3 for several HLS benchmarks reflect this fact. Our method and the one reported in (Karfa, 2007) take equal number of iterations and perform the same number of path extensions in all the cases; however, our method takes relatively more time. The overhead, however, is within an acceptable limit. If we disable weak equivalent path finding option from our tool, then it behaves exactly the same as the method reported in (Karfa, 2007). This redundant property checking for uniform code motion is, however, unavoidable because we could have combination of both uniform and non-uniform code motions in real verification cases. The implementations of the methods reported in (Kundu et al., 2010) and (Kim and Mansouri, 2008) and some of the benchmarks used in their paper are not available with us. Therefore, we have compared the time taken by our method with the same for the methods reported in (Kundu et al., 2010) and (Kim and Mansouri, 2008) based on the results available in their paper. The results are given as table 3.4. The time figures given in the table may differ marginally if the results are taken in the same system. The results in the table suggest that our method takes relatively less amount of time compared to other methods.

In our third experiment, we take the original behaviours and manually inject some errors in the behaviour. We then check the validity of such code motions by our equivalence checker. The objective of this experiment is to check the efficiency of our method in detecting incorrect code transformations. We have carried out several such experiments on each of the benchmarks. Some instances of such experiments are reported here. Other experiments also provide similar results. We have introduced the following code transformations: (i) Swapping two operations randomly (in DHRC

Errors	*Benchmarks*	*#opn*	*#iter*	*#extend*	*#modelChk*	*#time(s)*
type 1	DIFFEQ	2	5	1	2	0.342
	DHRC	3	30	0	0	0.200
type 2	GCD	1	11	3	4	0.175
	MODN	2	18	3	20	0.498
type 3	TLC	2	27	7	2	1.268
	IEEE754	4	79	25	40	107.243
type 4	BARCODE	4	114	62	98	21.208
	IEEE754	8	32	19	20	40.120

Table 3.5: Results for several high-level synthesis benchmarks on erroneous design

and DIFFEQ) which changes the data dependencies among the operations. (ii) non-uniform boosting up code motion which introduces false-data dependency in the other branch (in GCD and MODN) (iii) non-uniform duplicating down code motion which removes data dependency in the other branch (in TLC and IEEE754). (iv) mix of some correct code motions and incorrect code motions (in IEEE754 and BARCODE). The number of operations moved, the number of iterations, the number path extensions, the number of calls to model checker and the average execution time for this experiment are tabulated in table 3.5. For example, in the MODN (i.e., $(a*b)\%N$) behaviour, an operation $\mathtt{a} \Leftarrow \mathtt{a}/2$ is moved from its original place and is placed before its preceding if-else block. As a result, another operation $\mathtt{s} \Leftarrow \mathtt{s}+\mathtt{a}$ within the if-else block gets the wrong value of a. One of the the advantages of our method is that it can localize the source of non-equivalence and can report a specific path as a cause of that. For example, our method can report the path in which the value of s has differed in the MODN example. It is evident from table 3.5 that the worst case scenario (of complexity analysis) is not encountered in the errorenous cases also and our method finds the possible non-equivalence for the above examples quite efficiently. We also consider the bugs that have already been identified in practical compilers by the methods reported in (Gesellensetter et al., 2008) and in (Kundu et al., 2010). The objective of this study is to check whether such errors can also be detected by our method or not. The method reported in (Gesellensetter et al., 2008) identifies a bug in the *gcc* scheduler. The bug is due to violation of dependencies among certain operations. We consider a high-level version of the example given in (Gesellensetter et al., 2008) and run our tool on that example. Our method reports the non-equivalence of the behaviours as the data transformation of the paths do not match. However, we need an assembly level code to FSMD converter to apply our method on such application. Two bugs have been identified in the SPARK HLS tool by the method reported in (Kundu et al., 2010). One bug occurs in a particular corner case of copy propagation for array elements and the other bug is in the implementation of the code motion algorithm in the scheduler. Our current implementation does not support array. Therefore, we create an instance of the second bug of SPARK. This error is like the case (i) discussed above. We observe that our tool shows the non-equivalence of behaviours for this case.

3.7.1 Limitations of the method

Since the targeted verification problem is undecidable, the method presented here may produce some false negative results. Specifically, some of the limitations of our method are as follows:

In our path extension based equivalence of FSMDs, a path cannot be extended beyond loop boundaries as path extensions beyond loops are prevented by the definition of a path cover. As a result, the method fails when a code segment is moved beyond loops.

It may be recalled that we have considered the modulus and the division operations as functions in the normalized form. Therefore, if any synthesis tool deals with transformations over these operations, then normalization may fail to reduce them to syntactic identity; and hence our method fails to show the equivalence. Our normalization technique also fails for the case of operator strength reduction like replacing a product of two variables by repeated addition over a loop. More sophisticated normalization procedure needs to be evolved to handle such transformations.

Our method extends a path in the forward direction if the equivalent of that path is not found. This techniques fails to reveal the equivalence in the following case: Let there be a path β_1 followed by a path β_2 in FSMD M_0. Also, let there be a path α_1 followed by a path α_2 in FSMD M_1. Let the following equivalence between the paths of M_0 and M_1 hold: $\beta_1 \simeq \alpha_1$ and $\beta_2 \neq \alpha_2$ but $\beta_1\beta_2 \simeq \alpha_1\alpha_2$. Since $\beta_1\beta_2$ is equivalent to $\alpha_1\alpha_2$, the FSMDs are actually equivalent. Our method first finds $\beta_1 \simeq \alpha_1$. Since, equivalence of β_1 is found, our method does not extend β_1. In the next step, it finds the non-equivalence of β_2 and α_2 and reports a possible non-equivalence of the FSMDs.

An enhancement of our method with a heuristic of backward path extension can reveal this equivalence.

3.8 Conclusion

A novel equivalence checking method is presented in this chapter for verification of code motion transformations. The verification problem is treated as the equivalence checking problem of two FSMDs. The method is strong enough to handle both uniform and non-uniform code motions. For non-uniform code motions, it constructs specific data-flow properties automatically as CTL formulae and checks their satisfiability using a CTL model checker. The correctness and the complexity of the method are provided. The method is implemented and applied to validate the code motions used by a well known HLS tool called SPARK. The experiments show that the algorithm is usable for practical cases of equivalence preserving code motions.

Chapter 4

Verification of RTL generation phase of High-level Synthesis and RTL Transformations

4.1 Introduction

High-level synthesis (HLS) is the process of translating a behavioural description into a register transfer level (RTL) description containing a datapath and a controller (Gajski et al., 1992). The synthesis process consists of several sub-tasks carried out in sequence such as, scheduling, allocation and binding and datapath and controller (i.e., RTL) generation (Gajski et al., 1992). In the *datapath and controller generation* phase, the first task is to generate the datapath by providing a proper interconnection path from the source register(s) to the destination register for every RT-operation. The objective of this step is to maximize sharing of interconnection units among RT-operations ensuring conflict-free data transfers among the concurrent RT-operations. The second task is to generate the controller FSM by identifying the control signals required in each state. Such a synthesis flow is depicted in figure 4.1.

As discussed in subsection 2.2.2, a phase-wise verification technique that can handle the difficulties of each synthesis sub-task separately is desirable for HLS verification. A verification flow which works hand-in-hand with HLS is depicted in fig-

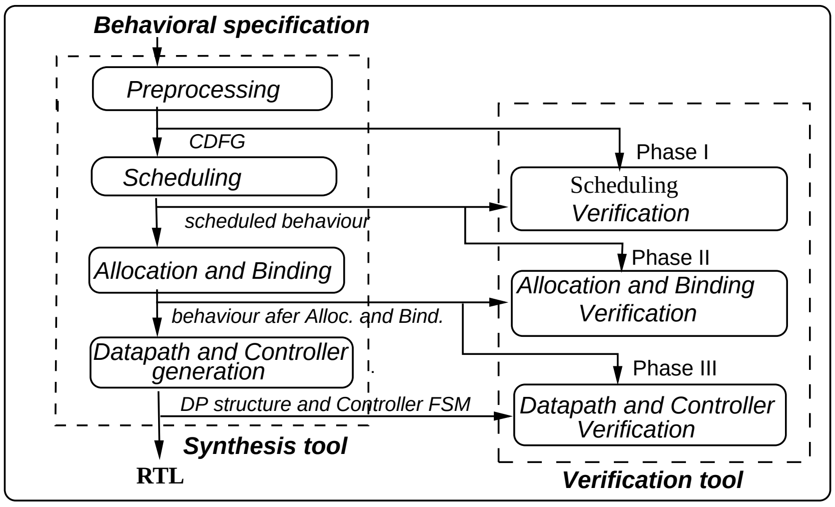


Figure 4.1: Hand-in-hand synthesis and verification framework

ure 4.1. A number of works were reported in the literature on verification of each phase of HLS. The methods proposed in Chapter 3 can be applied for verification of the scheduling phase. The verification of allocation and binding phase is treated in (Karfa, 2007; Mansouri and Vemuri, 1999). In this chapter, we present a verification method for the datapath and controller generation phase (i.e., RTL) assuming that the scheduling phase and the allocation and binding phase have already been verified.

In this work, verification of the RTL generation phase of HLS is accomplished in two steps as shown in figure 4.2. The input of this synthesis phase is modelled as an FSMD while the output comprises two parts, the datapath comprising the netlist and the controller represented as an FSM. In the first step of verification of this phase, an FSMD M_2 is constructed from the datapath interconnection information and the controller FSM. In the next step, equivalence between the FSMD M_1 representing the behaviour after the allocation and binding phase, and the FSMD M_2 is established. To verify RTL transformations, the inputs to our method are two RTL designs – the input RTL design and the one obtained from the input RTL by applying low power transformations. We construct FSMDs from both the input and the transformed RTLs using the same FSMD construction method as step 1 in figure 4.2 and then apply FSMD based equivalence checker.

In (Karfa, 2007), a preliminary version of this work has been reported. The contributions of this chapter over (Karfa, 2007) are as follows: 1. A rewriting based method for constructing an FSMD from a datapath and a controller description. The method

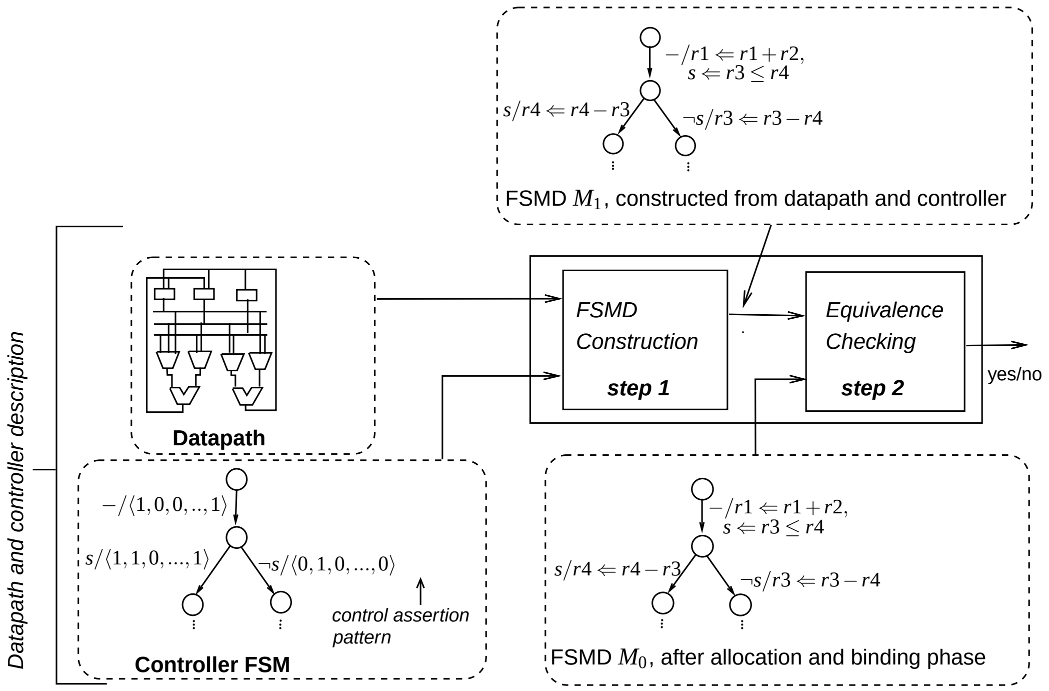

$-/r1 \Leftarrow r1+r2,$
$s \Leftarrow r3 \leq r4$
$s/r4 \Leftarrow r4-r3$
$\neg s/r3 \Leftarrow r3-r4$
FSMD M_1, constructed from datapath and controller
Datapath and controller description
Datapath
FSMD Construction step 1
Equivalence Checking step 2
yes/no
$-/\langle 1,0,0,..,1\rangle$
$s/\langle 1,1,0,...,1\rangle$
$\neg s/\langle 0,1,0,...,0\rangle$
control assertion pattern
Controller FSM
FSMD M_0, after allocation and binding phase

Figure 4.2: The steps of datapath and controller verification

is versatile enough to handle pipelined, multicycle and chained operations. 2. Rigorous treatments of soundness, completeness and complexity of the rewriting method. 3. Handling some algebraic transformations for interconnection optimization that occur during datapath synthesis using a normalization technique of arithmetic expressions. 4. We apply this method to verify RTL low power transformations. 5. An extensive experimental results are provided to show the effectiveness of the presented method.

The chapter is organized as follows. The challenges in verification of this phase are discussed in section 4.2. The basic issues involved in construction of the FSMD M_2 from the datapath and the controller FSM are discussed in section 4.3. The overall FSMD construction framework is given in section 4.4. Various flaws in the datapath and controller descriptions which get detected during the rewriting process are also discussed here. In section 4.5, the correctness and the complexity of the rewriting method are given. The equivalence checking method is given in section 4.6. The verification of several RTL low power transformations is given in section 4.7. Experimental results on several HLS benchmarks are given in section 4.8. The chapter is concluded in section 4.9.

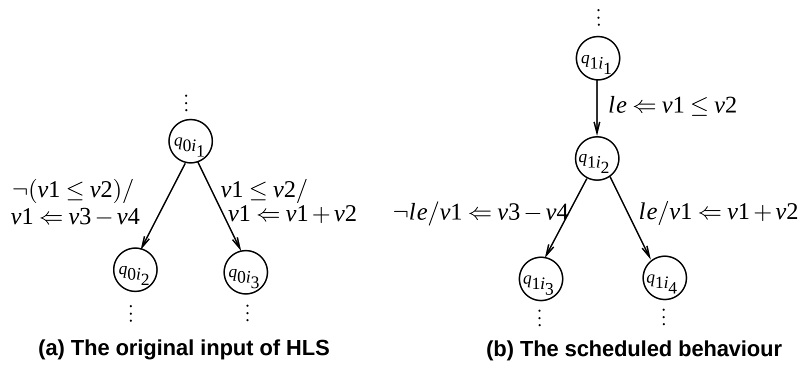


Figure 4.3: Scheduling of a relational operation

4.2 Verification challenges

Although RTL generation phase does not bring about any change in the control flow, the verification of this phase still has many challenges. First and foremost, the input RTL behaviour transforms to an output consisting of a datapath, which is merely a structural description, and a controller FSM. The controller FSM invokes a control assertion pattern (CAP), in each control step to execute all the required data-transfers and proper operations in the FUs. As a result, a set of arithmetic operations as well as a set of relational operations are performed in the datapath. To capture the computation of the condition of state transition in the initial behaviour, the scheduler introduces a set of Boolean variables B say, one for each relational operation, to store the result of that operation as depicted in figure 4.3 (where le is a Boolean variable). These Boolean variables (also called as status signals) are the inputs to the controller. The state transitions in the controller FSM are controlled by these Boolean variables. The verification task, therefore, involves identification of the RT-operations executed in a controller state from the control signal assertions in that state. The non-triviality of this task is due to the following reasons. First, it is not possible to obtain an RT-operation from a given control signal assertion pattern by examining the control signal values individually in isolation. This is because an RT-operation may involve a micro-operation which is accomplished by a set of control signals rather than an individual control signal. Secondly, each RT-operation is associated with a spatial sequence of micro-operations depicting the data flow from the source register(s) to a destination register. The analysis mechanism has to reveal this spatial sequence.

The second challenge in verification of this phase lies in handling multicyle or pipelined RT-operations which require more than one FSM state. For example, suppose an RT-operation involves a $k-$cycle functional unit (FU) in a state q of the input behaviour; then in the output behaviour, all the states in any path of length k leading to q should realize datapaths from the operand register(s) to the FU inputs while only state q should realize the datapath from the operand register(s) to the destination register. On the other hand, if the FU is a pipelined one, then only the $(k-1)^{th}$ predecessor state (and not the remaining ones) in any path leading to q should realize the operand datapaths. Accordingly, the control assertions in these states should reflect setting up of such partial datapaths. Thus, there may not be a *one-to-one correspondence between the control assertion pattern in an FSM state and the RT-operations in the corresponding state of the input behaviour*. We have not come across work on equivalence checking of pipelined or multicycle operations in the literature.

4.3 Construction of FSMDs from RTL designs

Let us now examine how the FSMD M_2 can be constructed from the datapath interconnection description and the controller FSM whose transitions are labelled with the subsets of B and the control assertion values. Construction of M_2 essentially consists in replacing the members of the subsets of B with the corresponding relational expressions over the DP registers and the control assertion values with the corresponding RT-operations. We shall describe the second task first and then discuss how the same method accomplishes the first task.

4.3.1 Representation of the datapath description

The following two pieces of information have to be extracted from the datapath description in order to find the RT-operations in each state of the FSMD M_2:

(i) *The set of all possible micro-operations in the datapath* - Let this set be denoted as $\mathscr{M}$. A data movement from an input y of a datapath component to its output x is encoded by the micro-operation $x \Leftarrow y$. The datapath components essentially are the storage elements (registers), the functional units, the interconnection components

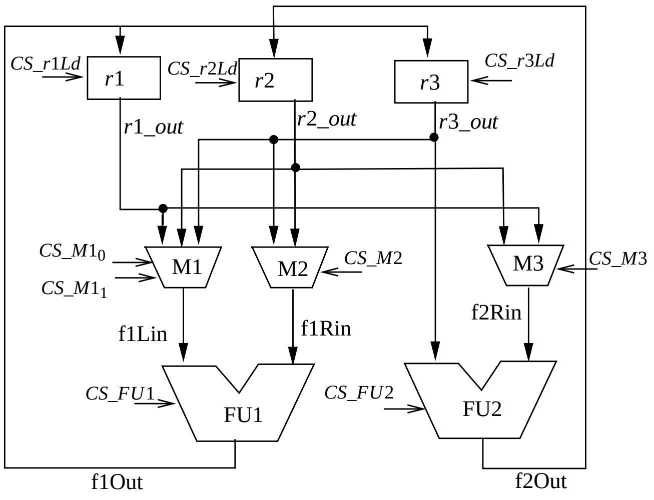


Figure 4.4: Datapath with control signals

(buses, muxes, de-muxes, switches, etc.) or the signal lines.

(ii) *The control signal assertion pattern for every micro-operation in* $\mathscr{M}$ - Let there be n control signals. A control signal assertion pattern needed for any micro-operation is represented as an ordered n-tuple of the form $\langle u_1,\ u_2, \ldots,\ u_n\rangle$, where each $u_i,\ 1 \leq i \leq n$, represents the value of the control signal c_i from the domain $\{0,\ 1,\ X\}$; $u_i = X$ implies that the control signal c_i is not required (relevant) for a particular micro-operation. In addition, for the same micro-operation, there may be two different combinations of control signals. Hence, the association between the micro-operations of a datapath and their corresponding CAPs is conceived as a relation $f_{mc} \subseteq \mathscr{M} \times \mathscr{A}$ where $\mathscr{A}$ is the set of all possible control assertion patterns; the tuple $(\mu, \rho) \in f_{mc}$ when the CAP ρ is needed for the micro-operation μ. So, the relation f_{mc} captures the datapath structure, in its entirety; the DP interconnection is conveyed by common signal naming.

Example 5 Let us consider the datapath shown in figure 4.4. In this figure, $r1,\ r2$ and $r3$ are registers, $M1$, $M2$ and $M3$ are multiplexers, $FU1$ and $FU2$ are functional units and $r1\_out$, $r2\_out$, $r3\_out$, $f1Lin$, $f1Rin$, $f2Rin$, $f1Out$, $f2Out$ are interconnection wires. The control signal names start with CS.

Let the ordering of the control signals in a control signal assertion pattern be $CS\_M1_1 \prec CS\_M1_0 \prec CS\_M2 \prec CS\_M3 \prec CS\_FU1 \prec CS\_FU2 \prec CS\_r1Ld \prec CS\_r2Ld \prec CS\_r3Ld$. The micro-operations of this datapath and their corresponding CAPs are given in the first two columns of table 4.1 with the first column designating $\mathscr{M}$ and the second one designating $\mathscr{A}$. It may be noted from the table that f_{mc} is actually a function in this case since we have only one CAP corresponding to each micro-operation. Ignore the third column of the table for the time being. The f_{mc} can be obtained from the output of any HLS tool containing the RTL behaviour of each component used in the datapath. □

4.3.2 A Method of obtaining the micro-operations for a control assertion pattern

The next task is to obtain the set of micro-operations $\mathscr{M}_A$ ($\subseteq \mathscr{M}$) which are activated by a given control assertion pattern A. The following definition is in order.

Definition 11 Superposition of assertion patterns: *Let A_1 and A_2 be two arbitrary control signal assertion patterns. Let $\pi_i(A)$ denote the i-th projection of an assertion pattern A which is the asserted value u_i of the control signal c_i. The assertion pattern, $A_1\ \theta\ A_2$, obtained by superposition θ of A_1 and A_2, satisfies the following conditions. For all i,*

$$\begin{aligned}
\pi_i(A_1\ \theta\ A_2) &= \pi_i(A_1),\ \text{for}\ \pi_i(A_1) = \pi_i(A_2) \\
&= \pi_i(A_1),\ \text{for}\ \ \pi_i(A_1) \neq \pi_i(A_2) \wedge \pi_i(A_1) = X \\
&= U(\text{undefined}),\ \text{for}\ \pi_i(A_1) \neq \pi_i(A_2) \wedge \\
&\qquad\qquad \pi_i(A_1) \neq X
\end{aligned}$$

The set of micro-operations $\mathcal{M}_A \subseteq \mathcal{M}$ which are activated by a given CAP A can be obtained by $\mathcal{M}_A = \{\mu \mid \mu \in \mathcal{M} \wedge (\mu, \rho) \in f_{mc} \wedge \rho\ \theta\ A = \rho\}$, where θ is the the superposition of assertion two patterns. The superposition of the control assertion pattern(s) of each micro-operation and the pattern A is checked one by one to decide whether to include that particular micro-operation in $\mathscr{M}_A$ or not. It may be noted that even if a micro-operation μ is activated by more than one CAPs, only one of those CAPs becomes true for a given CAP from the controller. Let us consider the superposition of one of the CAPS for a micro-operation μ and a given control assertion pattern A. It may be noted that bits in A being outputs of the controller circuit cannot

Micro-operation (μ)	Control assertion pattern of μ (ρ) s.t $(\mu,\rho) \in f_{mc}$	$\rho\ \theta\ A$
r1_out ⇐ r1	$\langle \mathbf{X}, \mathbf{X}, \mathbf{X}, \mathbf{X}, \mathbf{X}, \mathbf{X}, \mathbf{X}, \mathbf{X}, \mathbf{X} \rangle$	$\langle \mathbf{X}, \mathbf{X}, \mathbf{X}, \mathbf{X}, \mathbf{X}, \mathbf{X}, \mathbf{X}, \mathbf{X}, \mathbf{X} \rangle$
r2_out ⇐ r2	$\langle \mathbf{X}, \mathbf{X}, \mathbf{X}, \mathbf{X}, \mathbf{X}, \mathbf{X}, \mathbf{X}, \mathbf{X}, \mathbf{X} \rangle$	$\langle \mathbf{X}, \mathbf{X}, \mathbf{X}, \mathbf{X}, \mathbf{X}, \mathbf{X}, \mathbf{X}, \mathbf{X}, \mathbf{X} \rangle$
r3_out ⇐ r3	$\langle \mathbf{X}, \mathbf{X}, \mathbf{X}, \mathbf{X}, \mathbf{X}, \mathbf{X}, \mathbf{X}, \mathbf{X}, \mathbf{X} \rangle$	$\langle \mathbf{X}, \mathbf{X}, \mathbf{X}, \mathbf{X}, \mathbf{X}, \mathbf{X}, \mathbf{X}, \mathbf{X}, \mathbf{X} \rangle$
$f1Lin \Leftarrow r1\_out$	$\langle 0, 0, X, X, X, X, X, X, X \rangle$	$\langle U, 0, X, X, X, X, X, X, X \rangle$
$f1Lin \Leftarrow r2\_out$	$\langle 0, 1, X, X, X, X, X, X, X \rangle$	$\langle U, U, X, X, X, X, X, X, X \rangle$
f1Lin ⇐ r3_out	$\langle \mathbf{1}, \mathbf{X}, \mathbf{X}, \mathbf{X}, \mathbf{X}, \mathbf{X}, \mathbf{X}, \mathbf{X}, \mathbf{X} \rangle$	$\langle \mathbf{1}, \mathbf{X}, \mathbf{X}, \mathbf{X}, \mathbf{X}, \mathbf{X}, \mathbf{X}, \mathbf{X}, \mathbf{X} \rangle$
f1Rin ⇐ r2_out	$\langle \mathbf{X}, \mathbf{X}, \mathbf{1}, \mathbf{X}, \mathbf{X}, \mathbf{X}, \mathbf{X}, \mathbf{X}, \mathbf{X} \rangle$	$\langle \mathbf{X}, \mathbf{X}, \mathbf{1}, \mathbf{X}, \mathbf{X}, \mathbf{X}, \mathbf{X}, \mathbf{X}, \mathbf{X} \rangle$
$f1Rin \Leftarrow r3\_out$	$\langle X, X, 0, X, X, X, X, X, X \rangle$	$\langle X, X, U, X, X, X, X, X, X \rangle$
f2Rin ⇐ r2_out	$\langle \mathbf{X}, \mathbf{X}, \mathbf{X}, \mathbf{0}, \mathbf{X}, \mathbf{X}, \mathbf{X}, \mathbf{X}, \mathbf{X} \rangle$	$\langle \mathbf{X}, \mathbf{X}, \mathbf{X}, \mathbf{0}, \mathbf{X}, \mathbf{X}, \mathbf{X}, \mathbf{X}, \mathbf{X} \rangle$
$f2Rin \Leftarrow r1\_out$	$\langle X, X, X, 1, X, X, X, X, X \rangle$	$\langle X, X, X, U, X, X, X, X, X \rangle$
$f1Out \Leftarrow f1Lin + f1Rin$	$\langle X, X, X, X, 0, X, X, X, X \rangle$	$\langle X, X, X, X, U, X, X, X, X \rangle$
f1Out ⇐ f1Lin − f1Rin	$\langle \mathbf{X}, \mathbf{X}, \mathbf{X}, \mathbf{X}, \mathbf{1}, \mathbf{X}, \mathbf{X}, \mathbf{X}, \mathbf{X} \rangle$	$\langle \mathbf{X}, \mathbf{X}, \mathbf{X}, \mathbf{X}, \mathbf{1}, \mathbf{X}, \mathbf{X}, \mathbf{X}, \mathbf{X} \rangle$
f2Out ⇐ r3_out × f2Rin	$\langle \mathbf{X}, \mathbf{X}, \mathbf{X}, \mathbf{X}, \mathbf{X}, \mathbf{0}, \mathbf{X}, \mathbf{X}, \mathbf{X} \rangle$	$\langle \mathbf{X}, \mathbf{X}, \mathbf{X}, \mathbf{X}, \mathbf{X}, \mathbf{0}, \mathbf{X}, \mathbf{X}, \mathbf{X} \rangle$
$f2Out \Leftarrow r3\_out / f2Rin$	$\langle X, X, X, X, X, 1, X, X, X \rangle$	$\langle X, X, X, X, X, U, X, X, X \rangle$
r1 ⇐ f1Out	$\langle \mathbf{X}, \mathbf{X}, \mathbf{X}, \mathbf{X}, \mathbf{X}, \mathbf{X}, \mathbf{1}, \mathbf{X}, \mathbf{X} \rangle$	$\langle \mathbf{X}, \mathbf{X}, \mathbf{X}, \mathbf{X}, \mathbf{X}, \mathbf{X}, \mathbf{1}, \mathbf{X}, \mathbf{X} \rangle$
$r3 \Leftarrow f1Out$	$\langle X, X, X, X, X, X, X, X, 1 \rangle$	$\langle X, X, X, X, X, X, X, X, U \rangle$
r2 ⇐ f2Out	$\langle \mathbf{X}, \mathbf{X}, \mathbf{X}, \mathbf{X}, \mathbf{X}, \mathbf{X}, \mathbf{X}, \mathbf{1}, \mathbf{X} \rangle$	$\langle \mathbf{X}, \mathbf{X}, \mathbf{X}, \mathbf{X}, \mathbf{X}, \mathbf{X}, \mathbf{X}, \mathbf{1}, \mathbf{X} \rangle$

Table 4.1: Construction of the set $\mathscr{M}_A$ from f_{mc} for the control assertion pattern $A = \langle 1,0,1,0,1,0,1,1,0 \rangle$

contain 'X'. Let ρ be a CAP such that $(\mu,\rho) \in f_{mc}$. If $\pi_i(\rho) = X$, then $\pi_i(\rho\ \theta\ A)$ is also X. Now, consider some j such that $\pi_j(\rho) = 0$ or 1. If μ is executed by A, then $\pi_j(\rho) = \pi_j(A)$. So, $\rho\ \theta\ A$ becomes ρ if μ is performed by the assertion pattern A. Since $\pi_i(\rho) \neq U$ for any μ and i, and $\pi_i(\rho\ \theta\ A) = U$ when $\pi_i(\rho) \neq X \neq \pi_i(A)$, $\mu \notin \mathscr{M}_A$ *iff* $\pi_i(\rho\ \theta\ A) = U$, for some i.

Example 6 For the datapath given in figure 4.4, let the control assertion pattern in a particular FSM state be $A = \langle 1,0,1,0,1,0,1,1,0 \rangle$. The selection process is tabulated in table 4.1 (column 3). The set $\mathscr{M}_A$ comprises those micro-operations which are marked bold in the table. It may be noted that $\mu \in \mathscr{M} - \mathscr{M}_A$ iff it contains at least one U in some component; otherwise it is in $\mathscr{M}_A$. □

It may be noted that the construction of $\mathscr{M}_A$ cannot be achieved by examining each individual control signal value in A in isolation because a micro-operation may be accomplished by a set of control signals rather than an individual control signal. There is no information available in an assertion pattern to group the control signals so that each group defines a micro-operation around a datapath component.

4.3.3 Identification of RT operations realized by a set of micro-operations

Each RT-operation is accomplished by a set of concurrent micro-operations. For example, let us assume that an FU performs addition operation on its two input data $f1Lin$ and $f1Rin$. An RT-operation $r_3 \Leftarrow r_1 + r_2$ may be accomplished over a datapath by the concurrent micro-operations $r1\_out \Leftarrow r_1$, $r2\_out \Leftarrow r_2$, $f1Lin \Leftarrow r1\_out$, $f1Rin \Leftarrow r2\_out$, $f1Out \Leftarrow f1Lin + f1Rin$, $r3 \Leftarrow f1Out$. So, in order to find the concurrent RT-operations accomplished by a control assertion pattern A, it is necessary to find the RT-operations realized by the set $\mathscr{M}_A$ of concurrent micro-operations.

Finding an RT-operation from a given set of micro-operations is also not trivial because of two reasons. First, there may be more than one RT-operation realized in a state of the FSM. Secondly, there is a *spatial sequence* of concurrent micro-operations needed to accomplish an RT-operation but these are available in an unordered manner in $\mathscr{M}_A$.

The concurrent RT-operations accomplished by the set $\mathscr{M}_A$ of micro-operations are identified using a *rewriting method*. The method also reveals the spatial sequence of data flow needed for an RT-operation in a reverse order (from the destination register back to the source registers). The basic method consists in rewriting terms one after another in an expression. Let $\mathscr{M}'_A$ be the subset of $\mathscr{M}_A$ that contains all the micro-operations whose right hand side (RHS) expressions are to be rewritten. How the subset $\mathscr{M}'_A$ is chosen from $\mathscr{M}_A$ will be discussed shortly. For present discussion, it is sufficient to note that the set $\mathscr{M}'_A$ contains micro-operations of the form $r \Leftarrow r\_in$, where r is a register and r_in is its input terminal. Next, the RHS expression "r_in" is rewritten by looking for a micro-operation in $\mathscr{M}_A$ of the form "$r\_in \Leftarrow s$" or "$r\_in \Leftarrow s_1 \langle op \rangle s_2$". So, after rewriting "$r\_in$", we have the RHS expression, either of the form "s" or of the form "$s_1 \langle op \rangle s_2$". In the next step, s (or s_1 *and* s_2 for the latter case) are rewritten *provided they are not registers*. When the expression in hand is of the form "$s_1 \langle op \rangle s_2$" (and s_1, s_2 are not registers), rewriting takes place from left to right in a *depth-first manner*. Thus, at any point of time, the expression in hand can be of the form "$((s_1 \langle op_1 \rangle s_2) \langle op_2 \rangle s3) \langle op_3 \rangle_{\uparrow} \ldots$", where the pointer indicates the signal to be rewritten next and the signals s_1, s_2 and s_3 occurring at its left are all registers. The process terminates successfully when all s_i's in the expression in hand are registers.

Example 7 We illustrate the rewriting process for the datapath given in figure 4.4. Let us consider the control assertion pattern $A = \langle 1,0,1,0,1,0,1,1,0 \rangle$. Recall that the corresponding set $\mathscr{M}_A$ of micro-operations has been derived in example 6 as

$\{$ $r1\_out \Leftarrow r1$, $r2\_out \Leftarrow r2$, $r3\_out \Leftarrow r3$, $f1Lin \Leftarrow r3\_out$, $f1Rin \Leftarrow r2\_out$, $f2Rin \Leftarrow r2\_out$, $f1Out \Leftarrow f1Lin - f1Rin$, $f2Out \Leftarrow r3\_out \times f2Rin$, $r1 \Leftarrow f1Out$, $r2 \Leftarrow f2Out\}$.

The micro-operations in which a register occurs in the left hand side (LHS) are $r1 \Leftarrow f1Out$ and $r2 \Leftarrow f2Out$ which form the set $\mathscr{M}'_A$. The sequence of rewriting steps for the micro-operation $r1 \Leftarrow f1Out$ is as follows:

$r1 \Leftarrow f1Out$
$\Leftarrow f1Lin - f1Rin$ [by $f1Out \Leftarrow f1Lin - f1Rin$] $(step\ 1)$
$\Leftarrow r3\_out - f1Rin$ [by $f1Lin \Leftarrow r3\_out$] $(step\ 2)$
$\Leftarrow r3 - f1Rin$ [by $r3\_out \Leftarrow r3$ $(step\ 3)$
$\Leftarrow r3 - r2\_out$ [by $f1Rin \Leftarrow r2\_out$] $(step\ 4)$
$\Leftarrow r3 - r2$ [by $r2\_out \Leftarrow r2$] $(step\ 5)$

Similarly, the RT-operation $r2 \Leftarrow r3 \times r2$ can be obtained starting from the other micro-operation $r2 \Leftarrow f2Out$ in $\mathscr{M}'_A$. So, the RT-operations $r1 \Leftarrow r3 - r2$ and $r2 \Leftarrow r3 \times r2$ are executed by the given control assertion pattern A in a transition of the FSM. The forward spatial sequence of the micro-operations for an RT-operation is the reverse order in which they are used in the above rewriting steps; more specifically, therefore, the forward sequence of $r1 \Leftarrow r3 - r2$ is $r2\_out \Leftarrow r2$, $f1Rin \Leftarrow r2\_out$, $r3\_out \Leftarrow r3$, $f1Lin \Leftarrow r3\_out$, $f1Out \Leftarrow f1Lin - f1Rin$, $r1 \Leftarrow f1Out$. □

Let us now examine how the condition of execution associated with a controller FSM transition is made to correspond to an arithmetic predicate over registers. For this purpose, let us recall figure 4.3. The Boolean variable le may be bound to a register during allocation and binding phase (in the case of Mealy machine implementation of the controller FSM). In such a case, an arithmetic predicate (relational operation) will be realized in the same way as an arithmetic operation. However, if it is scheduled in the state itself (in the case of Moore machine implementation of the controller FSM), the Boolean variable le need not be stored in a register; instead it may be made available only as a status signal line output from the datapath feeding to the controller. We account for this case by simply including in the set $\mathscr{M}'_A$ the micro-operations containing status signals in their LHS in addition to the micro-operations having registers in the LHS.

4.3.4 Multicycle, pipelined and chained operations

Functional units have different propagation delays depending upon the functions they are designed to perform. As a result, concurrently activated units with delays shorter than a clock cycle remain un-utilized in most part of the clock cycle. To circumvent this problem, three well known techniques namely, *multicycle execution, pipelined execution and operation chaining* (Gajski et al., 1992) are used. In multicycle execution, the clock cycle is shortened to allow the fast operations to execute in one clock cycle, and the slower operations are permitted to take multiple clock cycles to complete execution as shown in figure 4.5(a). The corresponding figure 4.5(d) depicts the time steps for a k-cycle operation. Pipelining several sets of operands over the cycles of a multicycle execution of an operation p allows concurrent execution of p on each of those sets of operands (figure 4.5(b) and figure 4.5 (e)). Operation chaining allows two or more faster operations to be performed serially within one step (figures 4.5(c) and 4.5(f)). The datapath may have all such variations of FUs. Also, the controller needs to assert proper values to the control signals to execute an operation in a multicycle or pipelined way over multiple clocks or to execute more than one operation in one clock in a chained manner. Since, operation chaining is restricted to a single control state, the rewriting mechanism described in the previous subsection can be applied straightway. In the following, verification issues of other two cases are discussed.

To execute an operation in a multicycle FU, the data have to be held constant on the FU inputs over all the control steps it takes to execute that operation. In figure 4.5(d), for example, the inputs x_1 and y_1 are held on the FU inputs for k steps, where the FU is a k-cycle multiplier. If the operation starts execution at the i^{th} control step, then the result $x \times y$ is available only at the $(i+k-1)^{th}$ control step. Therefore, in order to ensure that the datapaths are set properly, we need to verify the following:

1. From i to $(i+k-1)$ steps, the left and the right input expressions to the FU are the same.

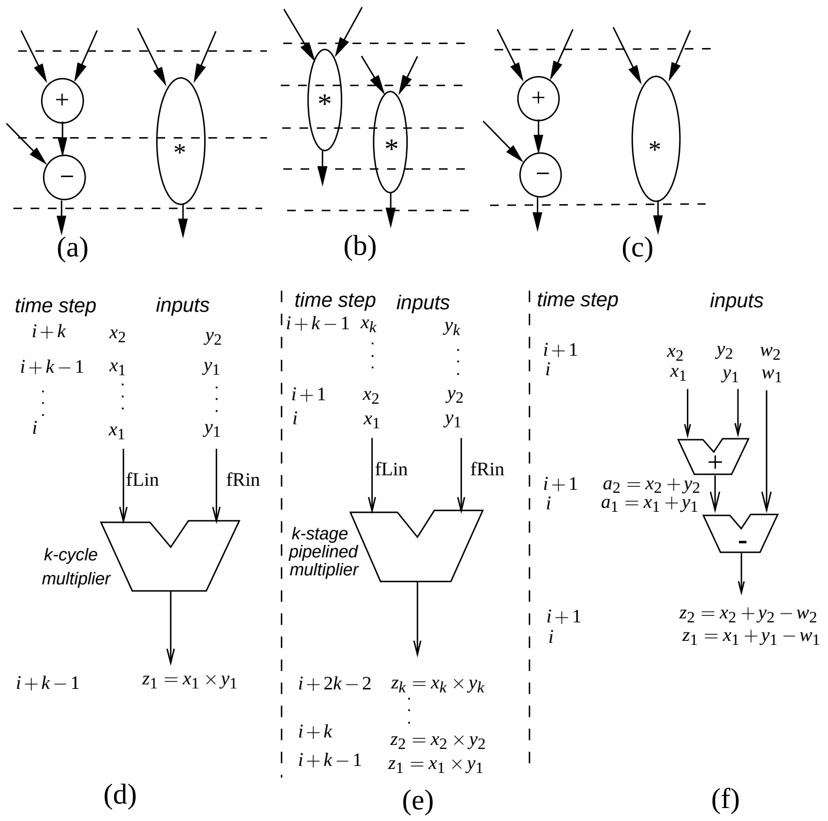


Figure 4.5: (a). schedule with a 2-cycle multiplier; (b) schedule with a 3-stage pipelined multiplier; (c) schedule with a chained adder and subtracter; (d) Input and output timing of a k-cycle multiplier; (e) Input and output timing of a k-stage pipelined multiplier; (f) Input and output timing for a chained adder and subtracter

2. The registers feeding the inputs to the FU are not updated (by any other operations) in the i^{th} to the $(i+k-1)^{th}$ time steps.

For a k-stage pipelined FU operation starting at the i^{th} step, the output of the operation is available at $(i+k-1)^{th}$ step. Figure 4.5(e) reflects the scenario. The FU, however, may take a new set of inputs in each of the $(i+1)^{th}$ step to the $(i+k-1)^{th}$ step as shown in the figure. So, the datapaths from the operand registers to the inputs of the FU are set in the i^{th} step, whereas the datapath from the output of that FU to the destination register where the result needs to be stored is set only at the $(i+k-1)^{th}$ step.

Based on the above discussion, in table 4.1 corresponding to f_{mc}, we need to record two additional pieces of information namely, the type and the cycle information of

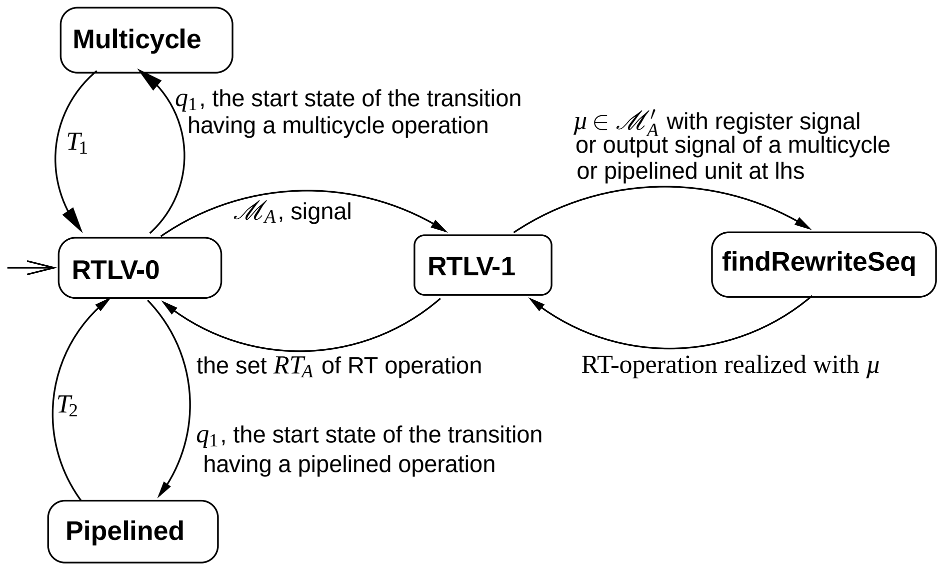

Multicycle
T_1
q_1, the start state of the transition having a multicycle operation
$\mu \in \mathscr{M}'_A$ with register signal or output signal of a multicycle or pipelined unit at lhs
$\mathscr{M}_A$, signal
RTLV-0
RTLV-1
findRewriteSeq
the set RT_A of RT operation
RT-operation realized with μ
T_2
q_1, the start state of the transition having a pipelined operation
Pipelined

Figure 4.6: Schematic of our FSMD construction framework from the datapath description and the controller FSM

each micro-operation to handle multicycle and pipelined operations. Let there be an FU performing a multicycle (pipelined) operation op needing k cycles (stages). The corresponding micro-operation is $p \Leftarrow L\langle op \rangle R$, where p, L and R are respectively the output, the left input and the right input of the FU. For such a micro-operation, we designate 'M' ('P') as type and k as cycle. For all other micro-operations, the type will be 'N' and the cycle value will be one. They are all normal single cycle operations. So, f_{mc} is now of the form $f_{mc} \subseteq \mathscr{M} \times type \times cycle \times \mathscr{A}$, where $type = \{$'M', 'N', 'P' $\}$ and $cycle = \mathbb{N}$, the set of natural numbers.

4.4 The Overall construction framework of FSMD

Figure 4.6 depicts the schematic of the method presented for construction of FSMD M_2. The module *RTLV-0* is the central module which takes the datapath description in the form of relation f_{mc} and the controller FSM and constructs the FSMD M_2. For each transition of the controller FSM, *RTLV-0* obtains the set $\mathscr{M}_A$ of micro-operations activated by the control assertion A associated with the transition; the method used for this purpose is discussed in subsection 4.3.2. It then invokes *RTLV-1* with the parameters $\mathscr{M}_A$ and the variable *signal*, the latter having values $\{0,\ 1,\ 2\}$. The assignment *signal* $= 0$ is used to obtain the RT-operations involving only single cycle operations or an RT-operation at the last step of a multicycle or a pipelined operation; *signal* $= 1$ is used to obtain an RT-operation with the output of a multicycle FU at the LHS (corresponding to all the cycles of a multicyle operation except the last one); *signal* $= 2$ is used to obtain an RT-operation with the output of a pipelined FU at the LHS (corresponding to the first cycle of a pipelined operation). Note that for a k-stage pipelined operation, only the first stage (where signal = 2) and the last step (where signal = 0) are relevant for the operation. The module *RTLV-1* computes the set $\mathscr{M}'_A$ of micro-operations to be rewritten based on the *signal* value and then calls the function $findRewriteSeq$ for each member of $\mathscr{M}'_A$. Initially, *RTLV-0* invokes *RTLV-1* for each transition of the controller FSM with signal equal to zero. The function $findRewriteSeq$, in turn, identifies the RT-operation starting from that micro-operation based on the procedure discussed in subsection 4.3.3. The modules *RTLV-1* and the function $findRewriteSeq$ are given as Algorithm 5 and Algorithm 6, respectively. It may be noted that the function $findRewriteSeq$ is capable of identifying the RT-operations that involve registers and outputs of pipelined units at the LHS. The module *RTLV-0* uses the module *Multicycle* and *Pipelined* to handle the multicycle and pipelined operations, respectively, the details of which are discussed in the subsequent subsections.

4.4.1 Handling of multicycle operations

It may be recalled that for any RT-operation $r \Leftarrow r1\langle op\rangle r2$, where "*op*" is a multicycle operation, there is one transition, τ say, in which the RT expressions $r \Leftarrow p$, $p \Leftarrow L\langle op\rangle R$, $L \Leftarrow r1$ and $R \Leftarrow r2$ are all realized; furthermore, each of the sequences of transitions of length $k-1$ leading to the transition τ will realize $p \Leftarrow r1\langle op\rangle r2$ by having $p \Leftarrow L\langle op\rangle R$, $L \Leftarrow r1$ and $R \Leftarrow r2$ in each member of the sequence. In addition, these transition sequences do not realize any other RT-operation which has "*r1*" or "*r2*" as LHS terms. We verify these facts by the following steps. The call graph of the same is shown pictorially in figure 4.6.

1. Let τ be a transition from the state q_1. Let *RTLV-0* identify by using *RTLV-1* the RT-operation $\mu : r \Leftarrow r1\langle op\rangle r2$ where op is a multicycle operation in τ (designated by the type 'M' in $f_{mc}|_{\mu}$). *RTLV-0* passes q_1 to the routine *Multicycle*

Algorithm 5 *RTLV-1*

/* Finds the set of RT-operations accomplished by a given set of micro-operations.

Input: The set $\mathscr{M}_A$ of micro-operations for a given control assertion pattern A and *signal* value.

Output: The set RT_A of RT-operations accomplished by $\mathscr{M}_A$. */

Method:

```
 1: Let $RT_A$ be $\phi$;
 2: if $signal = 0$ then
 3:     $\mathscr{M}'_A = \{\mu | \mu \in \mathscr{M}_A$ and $\mu$ has a register or a status line in its LHS term $\}$;
 4: else if $signal = 1$ then
 5:     $\mathscr{M}'_A = \{\mu | \mu \in \mathscr{M}_A$ and $\mu$ has an output signal of any multicycle FU in its LHS term$\}$;
 6: else if $signal = 2$ then
 7:     $\mathscr{M}'_A = \{\mu | \mu \in \mathscr{M}_A$ and $\mu$ has an output signal of any pipelined FU in its LHS term$\}$;
 8: end if
 9: if more than one micro-operation in $\mathscr{M}'_A$ has the same register name in its LHS then
10:     Report (“Same register is updated by more than one micro-operation");
11: else
12:     for each $\mu$ in $\mathscr{M}'_A$ do
13:         $replaced = \phi$;
14:         $Seq[0] \Leftarrow \mu$;
15:         $\mu \leftarrow findRewriteSeq\ (\mu,\ \mathscr{M}_A,\ replaced,\ Seq,\ 1)$;
            /* “$\mu$" - initially a micro-operation which is finally transformed to an RT-operation by the
            function.
            “$replaced$" - a set of signals rewritten already - used by the function to detect if a data flow
            loop is set up by the control assertion.
            “$Seq$" contains the final sequence of micro-operations used in rewriting - depicts the data
            flow in reverse, which obtains the RT-operation.
            The last parameter contains the number of micro-operations currently in $Seq$ */
16:         $RT_A = RT_A \cup \{\mu\}$;
17:     end for
18: end if
```

Algorithm 6 $findRewriteSeq\ (\mu,\ \mathscr{M}_A,\ replaced,\ Seq,\ i)$

/* replaces (rewrites) the leftmost non-register signal(s) in the RHS expression of μ, if possible, using some micro-operation m; accordingly, puts s in $replaced$, m in Seq (as the i^{th} entry) and invokes itself recursively; finally returns the rewritten μ having only register signals at its RHS */

```
 1: if the RHS of the $\mu$ contains either register signals or the output signals of some pipelined FU then
 2:    Report (“the RT operation found is $\mu$"); return $\mu$;
          /* terminates successfully */
 3: else
 4:    Let $s$ be the leftmost non-register signal in the RHS expression of $\mu$ which is neither a register
       nor an output of a pipelined FU.
 5:    if $s \in replaced$ then
 6:       Report (“loop set up in the datapath by the control assertion"); return empty RT-operation;
 7:    else
 8:       Let $\mathscr{M}_s \subset \mathscr{M}_A$ be the set of micro-operations s.t. each member of $\mathscr{M}_s$ has $s$ as its LHS signal.

 9:       if $\mathscr{M}_s == \phi$ then
10:          Report (“Inadequate set of micro-operations"); return empty RT-operation;
             /* No micro-operation found in $\mathscr{M}_A$ which has $s$ as its LHS signal */
11:       else if $\mathscr{M}_s$ contains more than one micro-operation then
12:          Report (“data conflict"); return empty RT-operation
             /* more than one driver activated for a signal */
13:       else
14:          /* $\mathscr{M}_s$ contains a single micro-operation. */
             Let $\mathscr{M}_s$ be $\{m\}$;
15:          $Seq[i] = m$;
16:          Let $m$ be of the form $s \Leftarrow e$; Let $\mu$ be of the form $t \Leftarrow ((e_1)s(e_2))$;
17:          replace all the occurrences of $s$ in the RHS expression of $\mu$ with the RHS expression of $m$;
             thus $\mu$ becomes $t \Leftarrow ((e_1)(e)(e_2))$;
18:          $replaced = replaced \cup \{s\}$;
19:          return $findRewriteSeq(\mu,\ \mathscr{M}_A,\ replaced,\ Seq,\ i+1)$;
20:       end if
21:    end if
22: end if
```

along with the number k of clock cycles needed for μ (obtained from $f_{mc}|_{\mu}$). The latter carries out a backward BFS traversal over the control FSM from the state q_1 (with depth = 1) up to a depth of k to identify all the sequences of transitions of length $k-1$ which terminate in q_1. Each transition occurring in these sequences is to be checked for containing the RT-operation $p \Leftarrow r1\langle op\rangle r2$ subsequently by *RTLV-0*. So the routine *Multicycle* returns the set, T_1 say, of all these transitions.

2. On obtaining the set T_1, the module *RTLV-0* selects transitions from T_1 one by one for finding the RT-operations realized in them using *RTLV-1* with $signal = 1$. *RTLV-1* puts in $\mathscr{M}'_A$ those micro-operations which have at their LHS the outputs of multicycle FUs and subsequently ensures that the members of T_1 contain the operation $p \Leftarrow r1\langle op\rangle r2$.

3. The module *RTLV-0* now checks for a micro-operation which has "*r1*" or "*r2*" as its LHS term in each transition of the set T_1. If such a micro-operation is found by *RTLV-0*, it reports an error message indicating that an operand is disturbed during a multicycle operation.

Example 8 Let us consider the controller FSM given in figure 4.7. The control assertion pattern associated with each of FSM's transition is not shown explicitly for clarity. Let us assume that *RTLV-0* identifies by using *RTLV-1* the RT-operation $r1 \Leftarrow r2 \times r3$ which is realizable by the assertion pattern associated with the transition $\langle q_1, q_2\rangle$. Let '$\times$' be a three cycle multiplier. We have to now ensure that the RT-operation $fuOut \Leftarrow r2 \times r3$ is realizable by the assertion patterns associated with the transitions belong to the sequences of transitions of length 2 $(= 3-1)$ which terminate in q_1. The module *Multicycle* finds this set of transitions T_1. For this example, the transition set is $T_1 = \{\langle q_i, q_j\rangle, \langle q_j, q_1\rangle, \langle q_2, q_j\rangle\}$. Also, we have to ensure that the registers $r2$ and $r3$ are not updated in any of the transitions in T_1. Task 1 and task 2 are done in step 2 and step 3, respectively, as described above. □

4.4.2 Handling of pipelined operations

For any RT-operation $r \Leftarrow r1\langle op\rangle r2$, where "*op*" is a k-stage pipelined operation, there is one transition in which the RT expression $r \Leftarrow p$ is realized and the first member

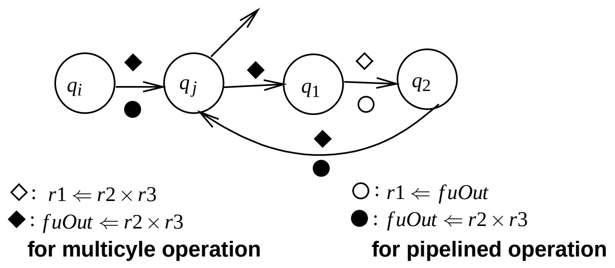

q_i
q_j
q_1
q_2
◇: $r1 \Leftarrow r2 \times r3$
◆: $fuOut \Leftarrow r2 \times r3$
for multicyle operation
○: $r1 \Leftarrow fuOut$
●: $fuOut \Leftarrow r2 \times r3$
for pipelined operation

Figure 4.7: Working with multicyle and pipelined operations

of each of the sequences of transitions of length $(k-1)$ leading to this transition will realize $L \Leftarrow r1$, $R \Leftarrow r2$ and $p \Leftarrow L\langle op\rangle R$. In other words, for an RT-operation $r \Leftarrow r1\langle op\rangle r2$ identified in the $(i+k-1)^{th}$ step, where $\langle op\rangle$ is a pipelined operation, the i^{th} step should contain the RT-operation $p \Leftarrow r1\langle op\rangle r2$ and the $(i+k-1)^{th}$ step should contain the RT-operation $r \Leftarrow p$. It may be noted that although p does not contain the value of the expression $r1\langle op\rangle r2$ in the i^{th} step, for convenience, we resort to such encoding to indicate that at the i^{th} step, the FU is activated to act on the operands "*r1*" and "*r2*". We ensure that the RT-operations are indeed obtained in the above manner by the following steps. The call graph of this sequence of steps is also shown pictorially in figure 4.6.

1. Let τ be a transition from the state q_1. Let *RTLV-0* identify by means of *RTLV-1* that the transition τ is one in which the RT-operation $\mu : r \Leftarrow p$ is realized, where p is the output of a pipelined FU (designated by the type 'P' in $f_{mc}|_{\mu}$). Now, *RTLV-0* passes q_1 to the routine *Pipelined* along with the number k of pipeline stages (obtained from $f_{mc}|_{\mu}$). The latter carries out a backward BFS traversal up to depth $k-1$ over the controller FSM from the state q_1 to identify all the sequences of transitions of length $k-1$ which terminate in q_1. The first member of all such transition sequences have to be checked for containing the RT-operation $p \Leftarrow r1\langle op\rangle r2$ subsequently by *RTLV-0*. So, the routine *Pipelined* returns the set, T_2 say, of all these first transitions in these sequence.

2. On obtaining the set T_2, the module *RTLV-0* selects transitions from T_2 one by one for finding the RT-operations realized in them using *RTLV-1* with the parameter *signal* $= 2$. For *signal* $= 2$, *RTLV-1* puts in $\mathscr{M}'_A$ only those micro-operations of $\mathscr{M}_A$ which have outputs of the pipelined FUs at their LHS. As *RTLV-1* invokes $findRewriteSeq$ with $\mu \in \mathscr{M}'_A$, the latter returns RT-operations

of the form $p \Leftarrow r1\langle op\rangle r2$. Thus, *RTLV-0* can ascertain that the members of T_2 indeed contain the desired RT-operation.

3. If it is found by *RTLV-0* that all the transitions in T_2 contain the operation $p \Leftarrow r1\langle op\rangle r2$, then it rewrites the RHS of the RT-operation $r \Leftarrow p$ (i.e., p) in the transition τ with the RHS expression of $p \Leftarrow r1\langle op\rangle r2$. So, finally the RT-operation $r \Leftarrow r1\langle op\rangle r2$ is associated with τ.

Example 9 Let us again consider the controller FSM given in figure 4.7. Let us assume that *RTLV-0* identifies by using *RTLV-1* the RT-operation $r1 \Leftarrow fuOut$ in transition $\langle q_1, q_2\rangle$. Let FU be a three stage pipelined multiplier. We have to now ensure that the RT-operation $fuOut \Leftarrow r2 \times r3$ is realizable by the assertion pattern associated with the first member of the sequences of transitions of length 2 $(= 3-1)$ which terminates in q_1. The module $Pipelined$ finds this set of transitions T_2. For this example, the transition set is $T_2 = \{\langle q_i, q_j\rangle, \langle q_2, q_j\rangle\}$. The above mentioned task is done by step 2. If step 2 is successful, then the actual RT-operation $r1 \Leftarrow r2 \times r3$ is obtained by rewriting $f2Out$ of $r2 \Leftarrow f2Out$ by the RHS expression of $f2Out \Leftarrow r3 \times r1$ in the transition $\langle q_1,\ q_2\rangle$ in step 3. □

4.4.3 Handling chained operations

The operation chaining scenario is depicted in figure 4.5(f) where two single cycle functional units are chained in the datapath. The results of both the FUs are available in the same time step as shown in the figure. As all the operations are performed in single cycle and there exists a spatial sequence among the operations that are in the chain, our rewriting method can handle this variation of the datapath. However, chaining of pipelined FUs, multicycle FUs with pipelined FUs, multicycle FUs with single cycle FUs and pipelined FUs with single cycle FUs are also possible in the datapath. Among them, the first two cases usually do not occur in practical circuits. The module *RTLV-0* can handle chaining of multicycle/pipelined FUs with single cycle FUs. The detailed implementation of *RTLV-0* is not given in this paper for brevity. It is also possible to extend our rewriting method to handle the first two scenarios of chaining.

4.4.4 Verification during construction of FSMD

Several inconsistencies that can be detected while constructing FSMD M_2 are as follows.

Loops set up in the datapath by the controller (steps 5 *and* 6 *of the function* $findRewriteSeq$*):* One non-register datapath signal line can be assigned only one value in a particular control step. If a non-register term is attempted to be rewritten twice during one invocation of $findRewriteSeq$ by *RTLV-1*, then it implies an improper control assertion pattern setting up a loop in the datapath without having any register.

Inadequate set of micro-operations performed by a control assertion pattern (steps 9 *and* 10 *of the function* $findRewriteSeq$*):* This situation arises due to either of the following two reasons:

(i) interconnection between two datapath components is not actually set by the control pattern but is required to complete an RT-operation or

(ii) the control signals are asserted in a wrong manner which leads to a situation where the required data transfer is not possible in the datapath.

Data conflict (steps 11 *and* 12 *of the function* $findRewriteSeq$*):* It means that data from more than one component try to pass through a single data line due to wrong control assertion.

Race condition (steps 9 *and* 10 *of RTLV-1):* It means that one register is attempted to be updated by two values in the same time step due to wrong control assertion pattern.

Error in the datapath for a multicycle operation: It occurs when the input paths for a k-cycle FU are not set in any of the first $k-1$ steps of execution of an operation in that FU. This occurs again due to wrong control assertion for the step in question and gets detected in *RTLV-0*.

Input operand is disturbed during execution of a multicycle operation: It means that an input register of a multicycle operation is updated by an RT-operation midway during execution of that multicycle operation. This situation arises due to either of the following two reasons: (i) If the RT-operation is also present in the FSMD M_1, then it is an error in the scheduling policy of the high-level synthesis, or (ii) If this RT-operation is not present in the FSMD M_1, then it occurs due to wrong control assertion which causes an erroneous RT-operation in the datapath. Such flaws get detected in *RTLV-0*.

Error in pipelining: It means that the datapaths corresponding to the inputs of a pipelined unit are not properly set due to wrong control assertion pattern at the state where the execution of that pipelined operation begins. This class of errors gets detected in *RTLV-0*.

4.5 Correctness and complexity of the algorithm

4.5.1 Correctness and complexity of the module *findRewriteSeq*

Theorem 5 *(Termination) The function $findRewriteSeq$ always terminates.*

Proof: If a recursive invocation does not detect one of the error situations depicted in steps 6, 10, 12 (and hence terminate), then it must replace a signal in the RHS expression of μ and enhance the set “replaced". The same signal, once replaced, is never replaced in subsequent invocations. There is only a finite number of signals in the datapath. Hence, the function cannot invoke itself more times than the number of signals in the datapath. $\square$

Definition 12 Forward rewriting by a micro-operation: *An expression e is said to be obtained from an expression e^- by forward rewriting by a micro-operation $s \Leftarrow e_r$, if e can be obtained by replacing one or more occurrences of e_r in e^- by s.*

In $findRewriteSeq$, the rewriting of an expression e_1 at hand by a micro-operation $s \Leftarrow e_2$ is carried out by replacing all the occurrences of s in e_1 by e_2. In contrast, the forward rewriting does the opposite, in keeping with the direction of data flow represented by the micro-operation (hence the name).

Lemma 3 *(*Realizability of an RT-operation*): An RT-operation $t \Leftarrow e$ is realizable over a datapath if there exists a sequence σ of micro-operations (over the datapath) such that the expression "t" is obtainable from the expression e by* forward rewriting *of e by the members of, and according to, the sequence σ.*

Proof: By induction on the length $|\sigma|$ of σ.

(Basis ($|\sigma| = 1$): σ comprises just one micro-operation, all micro-operations are realizable.

(Induction Step): Let us assume that whenever "t" is obtained from e by forward rewriting by a sequence of length less than i, the RT-operation $t \Leftarrow e$ is realizable.

Let us now assume that $\sigma = \langle \mu_0, \mu_1, \ldots, \mu_{i-1} \rangle$ obtains "t" from e by forward rewriting, where μ_k, $0 \le k \le i-1$, are micro-operations over the datapath. Let $\sigma = \langle \mu_0, \ldots, \mu_k,$ $\mu_{k+1}, \ldots, \mu_{i-1} \rangle$. Let e_k be obtained from e by forward rewriting of e by the subsequence $\sigma_k = \langle \mu_0, \mu_1, \ldots, \mu_k \rangle$. Obviously, "$t$" is obtained from e_k by the sequence $\langle \mu_{k+1}, \ldots, \mu_{i-1} \rangle$, In particular, for $k = 0$, therefore, "t" is obtained from e_0 by the sequence $\langle \mu_1, \ldots, \mu_{i-1} \rangle$ which is of length $i-1$. By induction hypothesis, the RT-operation $t \Leftarrow e_0$ is realizable over the datapath. Since, μ_0 is realizable and μ_0 obtains e_0 from e by forward rewriting, the RT-operation $t \Leftarrow e$ is realizable over the datapath and application of the micro-operations $\mu_0, \ldots \mu_{i-1}$ according to that sequence realizes it. $\square$

Theorem 6 *(Soundness): Let the function* findRewriteSeq *be invoked with a micro-operation μ of the form $t \Leftarrow e_0$ and the set $\mathscr{M}_A$ of micro-operations corresponding to the control assertion pattern A. Let it return an RT-operation p of the form $t \Leftarrow e$, where e comprises registers or the output signal of some pipelined FU. The RT-operation p is realizable over the datapath.*

Proof: Let the function terminate successfully (in step 1), obtain "Seq" as $\langle \mu_0, \mu_1, \cdots, \mu_k \rangle$ and return an RT-operation $t \Leftarrow e$. The LHS signal of the argument micro-operation $t \Leftarrow e_0$ is never disturbed by the function. Let us consider the reverse of “Seq" $\langle \mu_k, \cdots, \mu_1, \mu_0 \rangle = \sigma$, say. Thus, the first member μ_0 in “Seq" (that is, the last member in σ), is of the form $t \Leftarrow e_0$. Let the (RHS) expression obtained after application of μ_i in “Seq" be e_i. Clearly, $e_k = e$ and e contains registers or the output signal of pipelined FU(s). The fact that the expression “t" is obtainable from e_i by forward rewriting by the sequence $\langle \mu_i, \cdots, \mu_0 \rangle$, $0 \leq i \leq k$, can be proved by induction on i.

(*Basis* $i = 0$): “t" is obtainable from e_0 by forward rewriting by the singleton sequence of micro-operation(s) $\langle \mu_0 : t \Leftarrow e_0 \rangle$.
(*Induction Step*): Let “t" be obtainable from e_i by the forward rewriting by the sequence $\langle \mu_i, \cdots, \mu_1, \mu_0 \rangle$. Let e_i be of the form $x_1 s x_2$, where s is the leftmost signal in e_i such that s is neither a register nor an output signal of any pipelined FU. So from the steps 8 to 13 of the function, μ_{i+1} must be of the form $s \Leftarrow e_r$ and the function obtains e_{i+1} as $x_1 e_r x_2'$, where x_2' is obtained from x_2 by replacing all the occurrences of s in it by e_r. Thus, e_i is obtainable from e_{i+1} by forward rewriting by the singleton micro-operation sequence $\langle \mu_{i+1} \rangle$. From Induction hypothesis, “t" is obtainable from e_i by $\langle \mu_i, \ldots, \mu_0 \rangle$ and e_i is obtainable from e_{i+1} by μ_{i+1}. Therefore, “t" is obtainable from e_{i+1} by forward rewriting by the micro-operation sequence $\langle \mu_{i+1}, \mu_i, \cdots, \mu_0 \rangle$. This completes the induction.

Hence, in particular, t is obtainable from e_k (e) by forward rewriting by the sequence $\langle \mu_k, \mu_{k-1}, \cdots, \mu_0 \rangle$. From lemma 1, it follows that $t \Leftarrow e$ is realizable over the datapath by the sequence $\langle \mu_k, \mu_{k-1}, \cdots, \mu_0 \rangle$ (which is reverse of “Seq" found by the function). $\square$

In order to demonstrate the completeness of the rewrite procedure, we introduce the notion of *parse tree corresponding to a register transfer operation $t \Leftarrow e$ as realized by a given set $\mathscr{M}_A$ of micro-operations.* The parse tree of $t \Leftarrow e$ is the parse tree of the expression *e parenthesized in accordance with its realization by $\mathscr{M}_A$* with its root node labeled as t. For example, the RT-operation $t \Leftarrow r1 + r2 + r3$, realized over the datapath shown in figure 4.8(a), will have the parse tree corresponding to the expression $(r1 + r3) + r2$ (figure 4.8(b)) and not the one shown in figure 4.8(c) (corresponding to the expression $(r1 + r2) + r3$). From now onwards, we will leave the phrase “as realized by $\mathscr{M}_A$" understood following the term “parse tree of an RT-operation". We denote the parse tree of an RT-operation p as $T(p)$, and its depth as

$d(T(p))$; (the root is assumed to have depth 1). In general, there is a datapath signal, t_i say, associated with each non-leaf node of the parse sub-tree T_i of any sub-expression e_i of the RHS expression e. The sub-tree T_i (of e_i), therefore, is also the parse sub-tree corresponding to an RT-operation $t_i \Leftarrow e_i$.

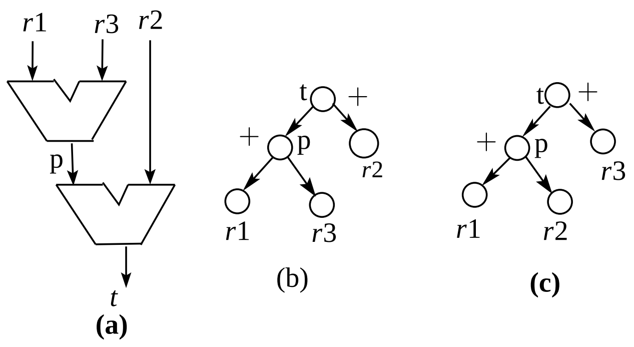


Figure 4.8: Parse tree of an RT-operation realized over a given datapath: An Example

Let e_i be of the form $e_{il}\langle op\rangle e_{ir}$ (i.e., the RT-operation is $t_i \Leftarrow e_{il}\langle op\rangle e_{ir}$), where “op" is a binary operation such as, ‘+’, ‘-’, ‘*’, ‘/’, etc.. The parse tree T_i of e_i has the root corresponding to the operator “op" (and the signal label t_i) and two subtrees namely, the left subtree T_{il} as the parse tree of e_{il} and the right subtree T_{ir} as the parse tree of e_{ir}. Here, the assignment operator $\Leftarrow$ is implicit with the operator $\langle op\rangle$. If e_i contains only e_{il}, then the parse tree T_i of e_i has only the signal label t_i associated with the root with $\Leftarrow$ kept implicit. Let t_{il} (t_{ir}) be the signal name attached to the root of T_{il} (T_{ir}). The *micro-operation sequence corresponding to the post-order traversal of* T_i is the sequence $\langle \sigma_{il}, \sigma_{ir}, t_i \Leftarrow t_{il}\langle op\rangle t_{ir}\rangle$, where σ_{il} (σ_{ir}) is the micro-operation sequence corresponding to the post-order traversal of T_{il} (T_{ir}). Similarly, the micro-operation sequence corresponding to the pre-order traversal of T_i is $\langle t_i \Leftarrow t_{il}\langle op\rangle t_{ir}, \sigma'_{il}, \sigma'_{ir}\rangle$, where σ'_{il} (σ'_{ir}) is the micro-operation sequence corresponding to the pre-order traversal of T_{il} (T_{ir}).

A sequence of micro-operations realizing an RT operation is only a spatial sequence of data flow and not a temporal sequence. A forward rewrite sequence realizing an RT operation $t_i \Leftarrow e_i$ can be presented as either the post-order traversal of its parse tree T_i or a minor variation of this order whereupon the orders of traversals of the subtrees are exchanged between themselves. We refer to this variant as *right-first post-order* traversal because the right subtree is traversed before the left subtree. The presence of non-commutative binary operations like ‘/’, ‘%’, etc., do not impair this fact. It may be noted that the pre-order traversal is the reverse of the right-first post-order traversal. Our completeness proof demonstrates that for any RT operation p realizable over the datapath (i.e., using micro-operations from $\mathscr{M}_A$), the function $findRewriteSeq$ produces the micro-operation sequence corresponding to the pre-order traversal of the parse tree of p or synonymously, the reverse of the sequence corresponding to the right-first post order traversal of the parse tree of p.

Let us define a *linear chain* over the datapath as an RT-operation of the form $d_l \Leftarrow d_r$, where d_l and d_r are any datapath signals.

Definition 13 Linear chain over the datapath: *A linear chain over the datapath is an RT-operation of the form $d_l \Leftarrow d_r$, where d_l and d_r are any datapath signals.*

For a linear chain which is realizable using a set $\mathscr{M}_A$ of micro-operations, there exists a micro-operation sequence $\sigma = \langle \mu_0, \mu_1, \cdots, \mu_k \rangle$, where $\mu_i \in \mathscr{M}_A$ and is of the form $d_{i+1} \Leftarrow d_i, 0 \leq i \leq k$, $d_0 = d_r$, $d_{k+1} = d_l$ and d_i's are the datapath signals such that d_l is obtained by forward rewriting by the members of σ. It might be noted that when $k = 0$, a linear chain is essentially a micro-operation. The parse tree of such a linear chain comprises only the root and is of depth one.

The micro-operation sequence realizing an operation $t \Leftarrow r1 \langle op \rangle r2$ may be viewed, in general, as $\langle \mu_0, \ldots, \mu_{j-1}, \mu_j, \ldots \mu_{i-1}, \mu_i, \mu_{i+1}, \ldots \mu_k \rangle$, where the subsequence $\langle \mu_0, \ldots,$ $\mu_{j-1} \rangle$ realizes a linear chain depicting data movement from $r2$ to the right input of the FU (typically of the form $fRin \Leftarrow r2$), the subsequence $\langle \mu_j, \ldots, \mu_{i-1} \rangle$ realizes a linear chain depicting data movement from $r1$ to the left input of the FU (typically of the form $fLin \Leftarrow r1$), μ_i is a micro-operation corresponding to the FU operation (typically of the form $fOut \Leftarrow fLin \ \langle op \rangle \ fRin$) and the subsequence $\langle \mu_{i+1}, \ldots, \mu_k \rangle$ realizes a linear chain of data movement from a functional unit (FU) output to the destination signal t.

Lemma 4 *For a realizable linear chain, the function $findRewriteSeq$ returns the reverse of the micro-operation sequence that realizes the linear chain.*

Proof: Let $d_l \Leftarrow d_r$ be a linear chain. Let the micro-operation sequence over the set $\mathscr{M}_A$ realizing the linear chain be $\sigma = \langle \mu_0, \mu_1, \cdots, \mu_k \rangle$. More specifically, let the corresponding forward rewriting sequence be $d_r \Rightarrow_{\mu_0} d_{r+1} \Rightarrow_{\mu_1} d_{r+2} \Rightarrow_{\mu_2} \ldots \Rightarrow_{\mu_{k-1}} d_{r+k}$

$\Rightarrow_{\mu_k} d_{r+k+1} = d_l$. It may be noted that $d_l \Leftarrow d_{r+k-i},\ 0 \leq i \leq k$, are all realizable linear chains realized by the forward rewriting micro-operation sequence $\langle \mu_{k-i}, \ldots, \mu_k \rangle$. It can be proved that for the realizable chain $d_l \Leftarrow d_{r+k-i}$, for any $i,\ 0 \leq i \leq k$, the function $findRewriteSeq$ obtains the sequence $\langle \mu_k, \mu_{k-1}, \ldots, \mu_{k-j} \rangle,\ 0 \leq j \leq i$, by induction on i.

Basis $[i = 0]$*:* $d_l \Leftarrow d_{r+k}$ is the micro-operation $\mu_k \in \mathscr{M}_A$ and the function finds $\langle \mu_k \rangle$ corresponding to this RT-operation (in step 8 and step 14.).

Induction Step: Let us assume that for $d_l \Leftarrow d_{r+k-i}$, the function returns $\langle \mu_k, \mu_{k-1}, \ldots,$ $\mu_{k-i} \rangle$. Let us now consider $d_l \Leftarrow d_{r+k-i-1}$. There is a forward rewriting sequence $d_{r+k-i-1} \Rightarrow_{\mu_{k-i-1}} d_{r+k-i} \Rightarrow_{\mu_{k-i}}, \ldots, \Rightarrow_{\mu_k} d_{r+k+1} = d_l$. Thus, $d_l \Leftarrow d_{r+k-i}$ is realizable and by induction hypothesis, the function first obtains d_{r+k-i} from d_l using the micro-operation sequence $\langle \mu_k, \ldots, \mu_{k-i} \rangle$ and then obtains $d_{r+k-i-1}$ from d_{r+k-i} by μ_{k-i-1}. Hence the result. □

The proof of completeness of the function consists in demonstrating that if the set $\mathscr{M}_A$ contains all the micro-operations needed for realizing an RT operation p, then the function $findRewriteSeq$ reveals in reverse order the sequence of micro-operations corresponding to the right-first post-order traversal of the parse tree of p thereby producing the sequence corresponding to the pre-order traversal of the parse tree of p.

Theorem 7 *(Completeness) If there is an RT-operation p of the form $t \Leftarrow e$ which is realizable using the micro-operations in $\mathscr{M}_A$, then the function $findRewriteSeq$, if invoked with a micro-operation of the form $t \Leftarrow e_0$, for some e_0, returns the sequence of micro-operations corresponding to the pre-order traversal of the parse tree of p, parenthesized according to its realization using $\mathscr{M}_A$.*

Proof: Let the sequence of micro-operations corresponding to the pre-order traversal of a tree of p be $\sigma_{pre}(p)$, that corresponding to the right-first-post-order traversal of p be $\sigma_{post}(p)$, and any sequence realizing p be $\sigma(p)$. Since $t \Leftarrow e$ is realizable using members of $\mathscr{M}_A$, there exists a sequence $\sigma_{post}(t \Leftarrow e)$ of the form $\langle \mu_0, \mu_1, \cdots, \mu_k \rangle$. Since σ is a spatial sequence and not a temporal one, without loss of generality, the suffix “post" can be used. We now prove that the function $findRewriteSeq$

returns the micro-operation sequence $\sigma_{pre}(p) = \langle \mu_k, \mu_{k-1}, \cdots, \mu_0 \rangle \ = \ reverse(\sigma_{post})$ We accomplish this proof *by induction on* $d(T(p)) \ = \ i$*, say*.

(Basis $i = 1$): The parse tree of p comprises the root labeled with t and e is another register or an output signal of a pipelined FU; thus, p is a realizable linear chain. By lemma 4, the micro-operation sequence $\langle \mu_k, \mu_{k-1}, \cdots, \mu_0 \rangle$ is obtainable by $findRewriteSeq$.

(Induction step): Suppose that the function can find the sequence σ_{pre}, when $d(T(p')) \leq i$, where p' is realizable and of the form $t \Leftarrow e$. Now, consider any realizable RT-operation p with $d(T(p)) = i+1$. Therefore, p must be of the form $t \Leftarrow e_1 \langle op \rangle e_2$, where $d(T(t_1 \Leftarrow e_1))$, $d(T(t_2 \Leftarrow e_2)) \leq i$. Let $\sigma_{post}(t_1 \Leftarrow e_1)$ and $\sigma_{post}(t_2 \Leftarrow e_2)$ be $\sigma_1 = \langle \mu_{1,0}, \mu_{1,1}, \ldots \mu_{1,l} \rangle$ and $\sigma_2 = \langle \mu_{2,0}, \mu_{2,1}, \ldots \mu_{2,r} \rangle$, respectively. The sequence $\sigma_{post}(p)$ is, therefore, $\langle \sigma_2,\ \sigma_1,\ \sigma_t \rangle$, where $\sigma_t = \sigma(t \Leftarrow t_1 \langle op \rangle t_2)$. So, the sequence $\sigma_{pre}(p) = reverse(\langle \sigma_2, \sigma_1, \sigma_t \rangle)$. Since, $d(T(t_1 \Leftarrow e_1))$, $d(T(t_2 \Leftarrow e_2)) \leq i$, by induction hypothesis, the function can construct the sequence $reverse(\sigma_1)$ and $reverse(\sigma_2)$. So, it remains to be proved that the function constructs (i) $reverse(\sigma_t)$ corresponding to $t \Leftarrow t_1 \langle op \rangle t_2$ and (ii) the sequence $\langle reverse(\sigma_t),\ reverse(\sigma_1), reverse(\sigma_2) \rangle = reverse(\langle \sigma_2, \sigma_1, \sigma_t \rangle)$ corresponding to p.

(i) The proof that the function constructs $reverse(\sigma_t)$ corresponding to $t \Leftarrow t_1 \langle op \rangle t_2$ is as follows: Let t_l, t_r and t_o respectively be the left input, the right input and the output of the FU which performs the operation "op". So, in order to realize the RT-operation $t \Leftarrow t_1 \langle op \rangle t_2$, it is necessary to realize the sequence of RT-operations $t_r \Leftarrow t_2$, $t_l \Leftarrow t_1$, $t_o \Leftarrow t_l \langle op \rangle t_r$, $t \Leftarrow t_o$, according to right-first postorder traversal of the parse tree of $t \Leftarrow t_1 \langle op \rangle t_2$. In other words, the realizing sequence σ_t of micro-operations can be split as follows: $\sigma_t = \langle \mu_0, \mu_1, \ldots, \mu_m \rangle = \langle \sigma_{t_2}, \sigma_{t_1}, \sigma_{t_3}, \sigma_{t_o} \rangle$, where $\sigma_{t_2} = \langle \mu_0, \mu_1, \ldots, \mu_{n_1} \rangle$ corresponds to (the parse tree of) the linear chain $t_r \Leftarrow t_2$, $\sigma_{t_1} = \langle \mu_{n_1+1}, \mu_{n_1+2}, \ldots, \mu_{n_1+n_2} \rangle$ corresponds to the linear chain $t_l \Leftarrow t_1$, σ_{t_3} is $\langle \mu_{n_1+n_2+1} = t_o \Leftarrow t_l \langle op \rangle t_r \rangle \rangle$ and $\sigma_{t_o} = \langle \mu_{n_1+n_2+2},\ \mu_{n_1+n_2+3}, \ldots, \mu_m \rangle$ corresponds to the linear chain $t \Leftarrow t_0$.

Now, the last micro-operation μ_m in the forward rewrite sequence must be of the form $t \Leftarrow t_i$, where t_i is the input signal name of the component whose output is t. Let the function $findRewriteSeq$ be invoked with μ_m as the argument. The function, in turn, selects its right hand side t_i for rewriting. Since $t \Leftarrow t_o$ is a linear chain realized by σ_{t_o}, by lemma 4, the function $findRewriteSeq$ constructs $reverse(\sigma_{t_o})$. In the process, the expression t_i changes to t_o. Therefore, $findRewriteSeq$ selects $\mu_{n_1+n_2+1}$ as the next micro-operation in the sequence and obtains the RT-operation as $t \Leftarrow t_l \langle op \rangle t_r$. The function next rewrites t_l to t_1, corresponding to the linear chain $t_l \Leftarrow t_1$; by lemma 4, therefore, the function constructs $reverse(\sigma_{t_1})$. By similar argument, the function then constructs $reverse(\sigma_{t_2})$ to rewrite t_r to t_2. Thus, the function constructs the sequence $\langle reverse(\sigma_{t_o}), \mu_{n_1+n_2+1}, reverse(\sigma_{t_1}), reverse(\sigma_{t_2}) \rangle = reverse(\langle \sigma_{t_2}, \sigma_{t_1}, \mu_{n_1+n_2+1}, \sigma_{t_o} \rangle)$ $= reverse(\sigma_t)$.

(ii) The proof that the function $findRewriteSeq$ constructs the sequence $\langle reverse(\sigma_t),$ $reverse(\sigma_1), reverse(\sigma_2) \rangle = reverse(\langle \sigma_2, \sigma_1, \sigma_t \rangle)$ for the RT-operation p is as follows: When $findRewriteSeq$ is invoked with $t \Leftarrow t_i$ in $\mathscr{M}'_A$, it returns the RT-operation $t \Leftarrow t_1 \langle op \rangle t_2$ by constructing the sequence $reverse(\sigma_t)$. The function $findRewriteSeq$ next takes up t_1 for rewriting. Since, p is realizable, so is the RT-operation $t_1 \Leftarrow e_1$ and, by assumption, $\sigma_1 = \sigma_{post}(t_1 \Leftarrow e_1)$. Since $d(T(t_1 \Leftarrow e_1)) \leq i$, by induction hypothesis, the function can construct $reverse(\sigma_1)$ to rewrite t_1 as e_1. Because of the strategy of replacing the leftmost non-register signal first, the function does rewrite t_1 as e_1 before taking up rewriting of t_2. Hence, the function obtains $t \Leftarrow e_1 \langle op \rangle t_2$ from $t \Leftarrow t_i$ $(= \mu_m \in \mathscr{M}'_A)$ constructing, in the process, the sequence $\langle reverse(\sigma_t), reverse(\sigma_1) \rangle$. It then takes up rewriting of t_2 to e_2; by similar argument as used above for rewriting of t_1 to e_1, it can be seen that the function constructs the sequence $reverse(\sigma_2)$ in course of this rewriting. Hence, it does construct $\langle reverse(\sigma_t), reverse(\sigma_1), reverse(\sigma_2) \rangle$ in rewriting $t \Leftarrow t_i$ to $t \Leftarrow e_1 \langle op \rangle e_2 = p$. $\square$

Complexity of $findRewriteSeq$ Let the number of FUs be f, the number of registers be r and the number of wires be w. Let the number of interconnect components (like muxes, demuxes, switches, etc.) be c and the maximum of number of inputs of an interconnect component be k. A micro-operation in the datapath involves either two wires through some interconnect unit or a register or an FU. So, the maximum number of micro-operations possible in the datapath is $O(kc + r + f)$. In each invocation of the function $findRewriteSeq$, one term (wire) is rewritten and no term is rewritten more than once. Hence, the number of invocations of the (recursive) function is $O(w)$. The maximum number of terms that can be present in an RHS expression is $O(r + w)$. So, the complexity of ensuring that no non-register signal is present in an RHS expression (i.e., the step 1 of $findRewriteSeq$) is $O(r + w)$. Similarly, the complexity of finding the leftmost non-register signal in an RHS expression (i.e., the step 4 of $findRewriteSeq$) and that of determining whether $s \in replaced$ (i.e., step 5 of $findRewriteSeq$) is $O(r+w)$. The maximum number of micro-operations in $\mathscr{M}_A$ is $O(c+r+f)$. Therefore, the complexity of finding a subset $\mathscr{M}_s$ of $\mathscr{M}_A$ (i.e. the step 8 of $findRewriteSeq$) is $O(c+r+f)$. So, the complexity of the function $findRewriteSeq$ is $O(w*((r+w)+(r+w)+(r+w)+(c+r+f))) = O(w^2+wr+wf+wc)$.

4.5.2 Correctness and complexity of the module *RTLV-1*

Theorem 8 *(Termination) The module RTLV-1 always terminates.*

Proof: The module *RTLV-1* constructs the set $\mathscr{M}'_A$ of micro-operations which have registers or an output signal of any multicycle or pipelined FU in lhs depending upon the value of signal it receives from *RTLV-1* and invokes the $findRewriteSeq$ function for each of its members. This set is obviously finite as a data path always consists of a finite number of micro-operations. Also, the function $findRewriteSeq$ always terminates. Hence, the module *RTLV-1* always terminates. □

Theorem 9 *(Soundness) If RTLV-1 terminates successfully returning a non-empty set* RT_A *of RT operations, then each member of* RT_A *is realizable by the assertion pattern A and is either of the form* $t \Leftarrow e$*, where t is a register signal or an output of a pipelined FU and e involves register signals or output signals of pipelined FUs or of the form* $p \Leftarrow e$*, where p is an output signal of a multicycle FU and e involves register signals.*

Proof: *RTLV-1* constructs the set $\mathscr{M}'_A$ consisting of micro-operations which have registers or output signals of multicycle or pipelined FUs in their lhs depending upon the value of *signal* it receives and invokes the function $findRewriteSeq$ for each member of the set $\mathscr{M}'_A$. Hence, the proof follows from the soundness of the function $findRewriteSeq$. □

Theorem 10 *(Completeness) If an RT-operation of the form $t \Leftarrow e$, where the rhs expression e contains registers and the output signal of pipelined FUs and the lhs signal t is either a register or the output signal of a pipelined FU or of the form $p \Leftarrow e$, where p is the output of a multicycle FU and the rhs expression contains only registers, is realizable by the set $\mathscr{M}_A$ of micro-operations for a given control assertion pattern A, then the RT operation will be found by RTLV-1.*

Proof: The completeness proof of the *RTLV-1* follows directly from the completeness of the function $findRewriteSeq$ (theorem 7). □

Complexity: It might be recalled from the section 4.5.1 that the complexity of the $findRewriteSeq$ function is $O(w^2 + wr + wf + wc)$, where w is the number of wires, f is the number of FUs, r is the number of registers and c is the number of interconnect components in the data path. The module *RTLV-1* invokes the $findRewriteSeq$ function for each member of the set $\mathscr{M}'_A$ of micro-operations which have in the lhs registers or output signals of functional units performing multicycle or pipelined operations. Therefore, the maximum number of micro-operations possible in $\mathscr{M}'_A$ is $O(r + f)$. Hence, the complexity of the module *RTLV-1* is $O((r + f) * (w^2 + wr + wf + wc)) = O(w^2r + w^2f + wr^2 + f^2w + wrf + wcr + wcf)$.

4.5.3 Correctness and complexity of the modules *Multicycle* and *Pipelined*

Both the modules *Multicycle* and *Pipelined* deploy a backward BFS traversal up to the depth $k-1$ from the argument FSM state q_1 of the controller. Hence we have the following theorems.

Theorem 11 *(Termination) The module Multicycle and the module Pipelined always terminate.*

Theorem 12 *(Soundness of Multicycle) If the module Multicycle terminates successfully, then each member of the set T_1 is a transition which occurs in some sequence of* *transitions of length $k-1$ which terminates in the controller FSM state q_1 passed as argument.*

Theorem 13 *(Completeness of Multicycle) If a transition occurs in some transition sequence of length $k-1$ which terminates in the state q_1 of the controller FSM, then the module Multicycle puts that transition in T_1.*

Theorem 14 *(Soundness of Pipelined) If the module Pipelined terminates successfully, then each member of T_2 is the first transition in some sequence of length $k-1$ which terminates in the state q_1 of the controller FSM.*

Theorem 15 *(Completeness of Pipelined) If a transition is the first one in some sequence of length $k-1$ which terminates in the state q_1 of controller FSM, then the module Pipelined puts in T_2.*

Complexity: Both the modules *Multicycle* and *Pipelined* have the same complexity. Only difference of the *Pipelined* module with *Multicycle* is that it stores only the first transition in each of the sequences of transitions of length $k-1$ which terminates in q_1 whereas *Multicycle* stores all the transitions of such sequences. Let the number of edges in the controller FSM be e. Since both the modules perform backward BFS, the complexity of both these modules is $O(e)$.

4.5.4 Correctness and complexity of the module *RTLV-0*

Theorem 16 *(Termination) The module RTLV-0 always terminates.*

Proof: For each transition, selected at random, the module *RTLV-0* first obtains the set $\mathscr{M}_A$ of micro-operations for the control assertion pattern A of that transition. As the number of micro-operations in the data path is finite, this step always terminates. The module *RTLV-0*, next, calls the module *RTLV-1* to find the set of RT operations performed by the control assertion pattern of that transition. As *RTLV-1* always terminates, this step also always terminates. If the set RT_A returned by *RTLV-1* contains an RT operation of the form $r \Leftarrow e'\langle e1\langle op\rangle e2\rangle e''$, where op is a multicycle operation, then *RTLV-0* calls *Multicycle* module to find the set T_1. This step obviously terminates. Specifically, for a multicycle operation, *RTLV-0* invokes *RTLV-1* to ensure that each transition in T_1 contains a desired RT operation. This step terminates because *RTLV-1* terminates. *RTLV-0* then ensures that none of the lhs of the micro-operations corresponding to a transition of the set T_1 contains a register signal which is present in the expressions $e1$ and $e2$. The finiteness of the set T_1 and that of the number of micro-operations in each transition confirm the termination of this step. For a pipelined operation of the form $r \Leftarrow e'pe''$, where p is the output of a pipelined FU, *RTLV-0* calls the function $Pipelined$ to find the set T_2. From termination of *Pipelined* follows the termination of this step. After that, it calls *RTLV-1* for each member of the set T_2. *RTLV-0* then ensures that all members of T_2 contain RT operations of the form $p \Leftarrow e$. The finiteness of this set and the termination of *RTLV-1* ensure that these two steps always terminate. □

Theorem 17 *(Soundness) If RTLV-0 terminates successfully, then the RT operations found by it corresponding to each transition are realizable over the data path by the control assertion pattern A associated with the transition.*

Proof: All single cycle, multicycle, pipelined and chained RT operations are essentially found by the function $findRewriteSeq$; the soundness of $findRewriteSeq$ ensures their realizability. The only extra RT operation that *RTLV-0* associates is a pipelined one in step 32 ensuring that (i) there is a transition, τ say, in which an RT operation of the form $r \Leftarrow p$, where p is the output of a pipelined FU with operation 'op', is realized and (ii) any transition that precedes τ by $(k-1)$ transitions has the RT operation $p \Leftarrow r1\langle op\rangle r2$ realized over the data path. These two conditions together, when met, ensure that $r \Leftarrow r1\langle op\rangle r2$ is realizable over the data path in transition τ.

□

Theorem 18 *(Completeness) If an RT operation is realizable over the data path by a control assertion pattern of a transition of the controller FSM, then the module RTLV-0 finds that RT operation.*

Proof: If the realizable RT operation in a transition, τ say, is a single cycle or a chained one, then *RTLV-0* finds it by $findRewriteSeq$. If it is a multicycle one, then also *RTLV-0* finds it by $findRewriteSeq$. The extra processing indulged in by *RTLV-0* is only to ensure certain conditions. Therefore, the completeness of $findRewriteSeq$ (and hence of *RTLV-1*) ensures that *RTLV-0* finds these RT operations. If it is a pipelined RT operation, then *RTLV-0* is ensured to find an RT operation of the form $r \Leftarrow e'pe''$, where p is the output of the corresponding pipelined FU, by the completeness of *RTLV-1*. It is then ensured to find the $(k-1)$-predecessors of τ (in T_2) using the module *Pipelined*. Finally, it is ensured to find in each of these $(k-1)$-predecessors in T_2, the RT operation $p \Leftarrow r1\langle op\rangle r2$, by the completeness of *RTLV-1*. □

Complexity: The module *RTLV-0* first obtains the set $\mathscr{M}_A$ of micro-operations for a control assertion pattern A using the mechanism described in section 4.3.2. It essentially compares the control assertion pattern of each micro-operation of the data path with A. The number of micro-operations possible in the data path is $O(kc+r+f)$. Let $|A|$ (i.e., the number of control signals over the entire data path) be n. So, the complexity of this step is $O(n*(kc+r+f))$. In the next step, the module *RTLV-0* invokes *RTLV-1* to find the RT operations realized by $\mathscr{M}_A$. So, the complexity of this step is $O(w^2r+w^2f+r^2w+f^2w+wrf+wcr+wcf)$. If any of the RT-operations returned by the module *RTLV-1* is of type $r \Leftarrow e'(e1\langle op_i\rangle e2)e''$ or $r \Leftarrow e'pe''$, it invokes either *Multicycle* for the former case to obtain the set T_1 or *Pipelined* for the latter one to obtain the set T_2. The complexity of both these modules is $O(e)$. The module *RTLV-0* now performs step 1 and step 2 as discussed above for each member of the set T_1 or T_2. The $|T_1|$ or $|T_2|$ is $O(e)$. So, the complexity of this step is $O(e*((knc+nr+nf)+(w^2r+w^2f+r^2w+f^2w+wrf+wcr+wcf)))$. The module searches for the RT operation $p \Leftarrow e1\langle op\rangle e2$ in the set T_1 for multicycle operations and $p \Leftarrow e$ on the transitions in T_2 for pipelined operations. Maximum number RT-operations, which has FU output signal in their lhs performed by the micro-operations for a control assertion pattern is $O(f)$. So, the complexity of this step of module *RTLV-0* is $O(f*e)$. Finally, it searches in $\mathscr{M}_{A_t}$ for a micro-operation which contains a register term belong to the expressions $e1$ or $e2$ on the transitions in T_1. So, the complexity of this step is $O(e*(r*(c+r+f)))$. Hence, the overall complexity of the module *RTLV-0* is $O((nkc+nr+nf)+(w^2r+w^2f+r^2w+f^2w+wrf+wcr+wcf)+(e*((nkc+nr+nf)+(w^2r+w^2f+r^2w+f^2w+wrf+wcr+wcf)+f*e+e*(r*(c+r+f)))$

$= O(w^2re + w^2fe + r^2we + f^2we + enkc + ewrf + ewcr + enr + enf)$.

4.6 Verification by equivalence checking

In the datapath and controller generation phase, the behaviour represented by the input FSMD M_1 is mapped to hardware. The number of states and the control structure of the behaviour are not modified in this phase. Hence, there is a one-to-one correspondence between the states of the input FSMD M_1 and the constructed FSMD M_2. Let the mapping between the states of M_1 and those of M_2 be represented by a function $f_{12} : Q_1 \leftrightarrow Q_2$. The state q_{2i} ($\in Q_2$) of the FSMD M_2 is said to be the corresponding state of q_{1i} ($\in Q_1$) if $f_{12}(q_{1i}) = q_{2i}$. A transition $q_{2k} \xrightarrow[c]{} q_{2l}$ of the FSMD M_2 is said to correspond to a transition $q_{1k} \xrightarrow[c']{} q_{1l}$ of the FSMD M_1 if $f_{12}(q_{1k}) = q_{2k}$, $f_{12}(q_{1l}) = q_{2l}$ and the condition c is equivalent to the condition c'. A set of RT-operations are formed for each state transition of the FSMD M_2 from the corresponding control assertion pattern. Now, the question is whether all the RT-operations corresponding to each state transition of the FSMD M_1 are captured by the controller or not. It may be noted that because of minimization of the controller output functions, some spurious RT-operations may get activated. So, the verification tasks consist in showing that all the RT-operations in each state transition in FSMD M_1 are also present in the corresponding state transition in FSMD M_2 and no extra RT-operation occurs in that transition of the FSMD M_2.

It may be noted that algebraic transformation techniques based on commutativity, associativity and distributivity of arithmetic operations are often used during datapath synthesis to improve interconnection cost (Chandrakasan et al., 1995b; Zory and Coelho, 1998). Hence, the RTL operations in one transition of FSMD M_1 and in the corresponding transition of FSMD M_2 may not be syntactically identical. Specification of digital systems implementing algorithmic computations involves the whole of integer arithmetic for which a canonical form does not exist. Instead, we use a normal form adapted from (Sarkar and De Sarkar, 1989) during equivalence checking. The normalization process reduces many computationally equivalent formulas to a syntactically identical form. We have also added several simplification rules on normalized expressions, the details of which may be found in Chapter 3.

It has been assumed that the control flow graph of the behaviour is not changed subsequent to the scheduling phase. However, controller FSMs can be minimized (Bergamaschi et al., 1992). Under such a situation, our method, in its present form, will not be able to establish equivalence as the bijections no longer hold. If, however, the FSM minimization is a distinct phase following the datapath and controller generation phase, as is usually the case (Bergamaschi et al., 1992), then the verification of the FSM minimization can be separately addressed as the equivalence checking problem of two FSMs. Another way to upgrade the present verifier is to use the path based equivalence checking method as reported in Chapter 3; this approach, however, would have an exponential upper bound.

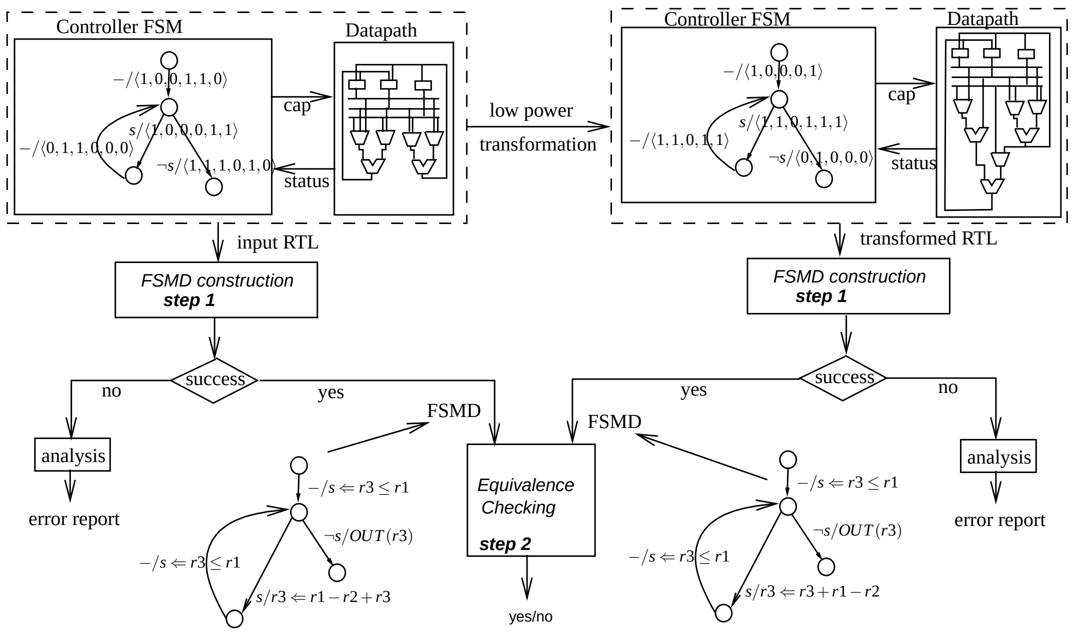

Controller FSM
Datapath
$-/\langle 1,0,0,1,1,0\rangle$
$s/\langle 1,0,0,0,1,1\rangle$
$-/\langle 0,1,1,0,0,0\rangle$
$\neg s/\langle 1,1,1,0,1,0\rangle$
cap
status
low power
transformation
Controller FSM
Datapath
$-/\langle 1,0,0,0,1\rangle$
$s/\langle 1,1,0,1,1,1\rangle$
$-/\langle 1,1,0,1,1\rangle$
$\neg s/\langle 0,1,0,0,0\rangle$
cap
status
input RTL
transformed RTL
FSMD construction step 1
FSMD construction step 1
success
success
no
yes
yes
no
analysis
analysis
error report
error report
FSMD
FSMD
Equivalence Checking step 2
yes/no
$-/s \Leftarrow r3 \leq r1$
$\neg s/OUT(r3)$
$-/s \Leftarrow r3 \leq r1$
$s/r3 \Leftarrow r1 - r2 + r3$
$-/s \Leftarrow r3 \leq r1$
$\neg s/OUT(r3)$
$-/s \Leftarrow r3 \leq r1$
$s/r3 \Leftarrow r3 + r1 - r2$

Figure 4.9: Verification framework

4.7 Verification of low power RTL transformations

We present an automated verification method for low power transformations in RTL design. The inputs are two RTL designs – the original one and the one obtained by applying some low power transformations. Our verification framework is depicted in figure 4.9. Specifically, we apply the FSMD construction mechanism developed above (in section 4.4) to obtain the FSMDs from both the RTL designs. In the next step, the equivalence of two FSMDs are established.

A significant portion of the overall power consumption of an RTL design is due to propagation of glitches in both control and data parts of the circuit. Reduction of glitching power is achieved primarily by the following transformations of RTL circuits:

- Choosing an alternative datapath architecture
- Restructuring of multiplexer networks to eliminate glitchy control signals
- Restructuring of multiplexer networks to enhance data correlation
- Clocking of control signals
- Inserting delay elements in the datapath and the controller

We observed that the number of control signals and the datapath interconnections in the input and the output RTL descriptions may differ due to applications of these transformations. However, the number of control states in the controller FSM remains the same in both RTLs. We have already proved by theorem 7 that any realizable RT-operation in the datapath can be constructed by our FSMD construction mechanism (described in section 4.4). Therefore, by virtue of the completeness of the method, it is able to construct the FSMDs from both the RTL designs. In the following, we illustrate the above mentioned transformations with examples to show their effects on the input RTL circuits and then discuss how our FSMD construction mechanism works for them.

4.7.1 Alternative datapath architecture

Increase in resources does not always increase power consumption of a circuit. Alternative datapath architectures (even with more resource) may be available with low power consumption. The following example illustrates the fact.

Consider two RTL architectures shown in figure 4.10(a) and in figure 4.10(b). Both the circuit implement the same function: `if(CS_m) then` $z \Rightarrow$ `c + d else` $z \Rightarrow$ `a + b`. Let the control signal *CS_m* be the output of a comparator block. The circuit in figure 4.10(b) uses more resource than the circuit in figure 4.10(a) since the former

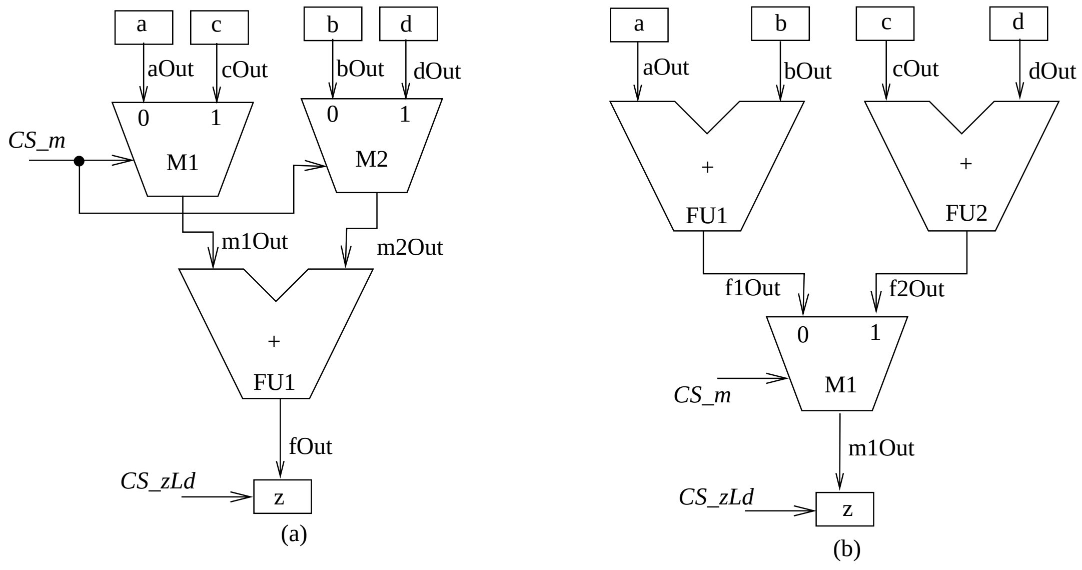


Figure 4.10: Alternative datapath architecture

one uses two adders as opposed to one adder for the latter. Since the control signal may come from a comparator circuit, it will be glitchy. In the circuit of figure 4.10(a), these glitches may then propagate through two multiplexers to the inputs of the adder, which cause a significant increase in glitching activities and hence power consumption in the two multiplexers and the adder. For the circuit in figure 4.10(b), though the comparator generates glitches as before, the effect of these glitches is restricted to the single multiplexer. Hence, the architecture in figure 4.10(b) actually consumes less power than the circuit in figure 4.10(a).

The controller's functionality would be the same even though the datapath architecture is changed. Therefore, the controller of this RTL shall generate the same control assertion pattern (CAP) for both the datapaths to perform certain RT-operation(s) in the datapaths. The existing rewriting method works for such transformations. Since the datapath architecture is changed, the micro-operations of the datapath are different. Therefore, the rewriting sequence would be different. The rewriting method finds the same RT-operations in both the datapaths for a given CAP. Following example illustrates this.

Let the relative ordering of the control signals in the CAP pattern be $CS\_m \prec CS\_zLd$. For brevity, let us ignore the other members in CAP. Let the controller generate the CAP $\langle 0, 1 \rangle$ in a control state. The possible micro-operations in the datapath

Micro-operation (μ)	CAP of μ	$f_{mc}(\mu)\ \theta\ A$
aOut ⇐ a	$\langle \mathbf{X}, \mathbf{X} \rangle$	$\langle \mathbf{X}, \mathbf{X} \rangle$
bOut ⇐ b	$\langle \mathbf{X}, \mathbf{X} \rangle$	$\langle \mathbf{X}, \mathbf{X} \rangle$
cOut ⇐ c	$\langle \mathbf{X}, \mathbf{X} \rangle$	$\langle \mathbf{X}, \mathbf{X} \rangle$
dOut ⇐ d	$\langle \mathbf{X}, \mathbf{X} \rangle$	$\langle \mathbf{X}, \mathbf{X} \rangle$
m1Out ⇐ aOut	$\langle \mathbf{0}, \mathbf{X} \rangle$	$\langle \mathbf{0}, \mathbf{X} \rangle$
m1Out ⇐ cOut	$\langle 1, X \rangle$	$\langle U, X \rangle$
m2Out ⇐ bOut	$\langle \mathbf{0}, \mathbf{X} \rangle$	$\langle \mathbf{0}, \mathbf{X} \rangle$
m2Out ⇐ dOut	$\langle 1, X \rangle$	$\langle U, X \rangle$
fOut ⇐ m1Out + m2Out	$\langle \mathbf{X}, \mathbf{X} \rangle$	$\langle \mathbf{X}, \mathbf{X} \rangle$
z ⇐ fOut	$\langle \mathbf{X}, \mathbf{1} \rangle$	$\langle \mathbf{X}, \mathbf{1} \rangle$

Table 4.2: The micro-operations of datapath in figure 4.10(b) and computation of $\mathcal{M}_{\mathcal{A}}$ for CAP $\langle 0,1 \rangle$

Micro-operation (μ)	CAP of μ	$f_{mc}(\mu)\ \theta\ A$
aOut ⇐ a	$\langle \mathbf{X}, \mathbf{X} \rangle$	$\langle \mathbf{X}, \mathbf{X} \rangle$
bOut ⇐ b	$\langle \mathbf{X}, \mathbf{X} \rangle$	$\langle \mathbf{X}, \mathbf{X} \rangle$
cOut ⇐ c	$\langle \mathbf{X}, \mathbf{X} \rangle$	$\langle \mathbf{X}, \mathbf{X} \rangle$
dOut ⇐ d	$\langle \mathbf{X}, \mathbf{X} \rangle$	$\langle \mathbf{X}, \mathbf{X} \rangle$
f1Out ⇐ aOut + bOut	$\langle \mathbf{X}, \mathbf{X} \rangle$	$\langle \mathbf{X}, \mathbf{X} \rangle$
f2Out ⇐ cOut + dOut	$\langle \mathbf{X}, \mathbf{X} \rangle$	$\langle \mathbf{X}, \mathbf{X} \rangle$
m1Out ⇐ f1Out	$\langle \mathbf{0}, \mathbf{X} \rangle$	$\langle \mathbf{0}, \mathbf{X} \rangle$
m1Out ⇐ f2Out	$\langle 1, X \rangle$	$\langle U, X \rangle$
z ⇐ m1Out	$\langle \mathbf{X}, \mathbf{1} \rangle$	$\langle \mathbf{X}, \mathbf{1} \rangle$

Table 4.3: The micro-operations of datapath in figure 4.10(b) and computation of $\mathcal{M}_{\mathcal{A}}$ for CAP $\langle 0,1 \rangle$

of figure 4.10(a) are: `aOut` ⇐ `a`, `bOut` ⇐ `b`, `cOut` ⇐ `c`, `dOut` ⇐ `d`, `m1Out` ⇐ `aOut`, `m1Out` ⇐ `cOut`, `m2Out` ⇐ `bOut`, `m2Out` ⇐ `dOut`, `fOut` ⇐ `m1Out` + `m2Out`, `z` ⇐ `fOut`. In figure 4.10(b), this set contains: `aOut` ⇐ `a`, `bOut` ⇐ `b`, `cOut` ⇐ `c`, `dOut` ⇐ `d`, `f1Out` ⇐ `a` + `b`, `f2Out` ⇐ `c` + `d`, `m1Out` ⇐ `f1Out`, `m1Out` ⇐ `f2Out`, `z` ⇐ `m1Out`. The selection of micro-operations that are activated by the CAP $\langle 0,1 \rangle$ in datapaths in figure 4.10(a) and in figure 4.10(a) are tabulated in table 4.2 and table 4.3, respectively. The selected micro-operations are marked as bold in the tables.

For the datapath in figure 4.10(a), the sequence of rewriting steps is as follows:

$z \Leftarrow fOut$
$\quad \Leftarrow m1Out + m2Out \quad$ [by $fOut \Leftarrow m1Out + m2Out$] $(step\ 1)$
$\quad \Leftarrow aOut + m2Out \quad$ [by `m1Out` ⇐ `aOut`] $(step\ 2)$

$\Leftarrow a + m2Out$ [by $\texttt{aOut} \Leftarrow \texttt{a}$] $(step\ 3)$

$\Leftarrow a + bOut$ [by $\texttt{m2Out} \Leftarrow \texttt{bOut}$] $(step\ 4)$

$\Leftarrow a + b$ [by $\texttt{bOut} \Leftarrow \texttt{b}$] $(step\ 5)$

For the datapath in figure 4.10(b), the sequence of rewriting step is as follows:

$z \Leftarrow m1Out$

$\Leftarrow f1Out$ [by $m1Out \Leftarrow f1Out$] $(step\ 1)$

$\Leftarrow aOut + bOut$ [by $\texttt{f1Out} \Leftarrow \texttt{aOut} + \texttt{bOut}$] $(step\ 2)$

$\Leftarrow a + bOut$ [by $\texttt{aOut} \Leftarrow \texttt{a}$] $(step\ 3)$

$\Leftarrow a + b$ [by $\texttt{bOut} \Leftarrow \texttt{b}$] $(step\ 4)$

4.7.2 Restructuring of multiplexer networks to enhance data correlation

The glitch propagation from control signals through a multiplexer is minimized when its data inputs are highly correlated (Raghunathan et al., 1999). This observation can be used to reduce the glitch propagation from control signals feeding a multiplexer network by restructuring it. The following example illustrate this fact.

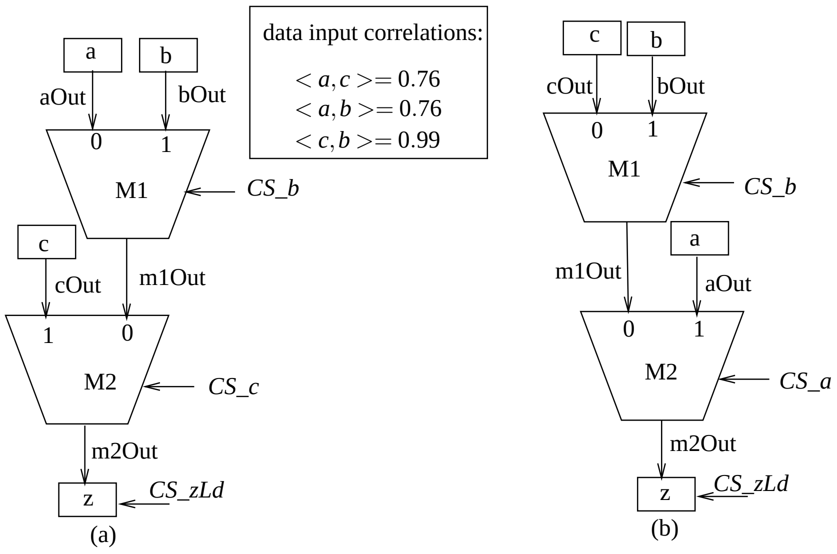


Figure 4.11: Multiplexer network restructuring to enhance data correlation: An example

Let us consider the 3 : 1 multiplexer network implementation using two 2 : 1 multiplexers in figure 4.11(a). In this example, at most one of *CS_a*, *CS_b* and *CS_c* can be 1 in a control step. Let us assume that these control signals are generated in such a way that *CS_b* be highly glitchy, leading to propagation of glitches to the output of $M1$. Let us also assume that data in c and b be highly corelated at bit-level as shown in the figure. Hence, the multiplexer tree is transformed as shown in figure 4.11(b). This arrangement significantly lowers the switching activity at the output of $M1$.

The datapath structure has changed. Therefore, the sets of micro-operations are different in the datapaths. However, the controller remains the same and it produces identical CAP for both the datapath to execute certain RT-operations. It may be noted that CS_a is redundant for the datapath in figure 4.11(a) whereas CS_c is redundant for the datapath in figure 4.11(b). In this case also, the existing rewriting method works. The rewriting sequence, however, differs for the datapaths. For a given CAP, the rewriting method obtains the same RT-operation(s) in both the datapaths.

4.7.3 Restructuring of multiplexer networks to eliminate glitchy control signals

The glitches of control signal propagates through the datapath; consequently, consumes a large portion of the glitching power of the entire circuit. Eliminating glitchy control signals, therefore, reduces the total power consumption of the circuit. Following example illustrates the fact.

Let us consider datapath in figure 4.12(a). The multiplexer network uses only two of the three available control signals. Let assume that the control signal *CS_a* be glitchy. An alternative implementation of the network is shown in figure 4.12(b) where the control signal *CS_a* is eliminated.

In the datapath in figure 4.12(a) the disjunction of control signals *CS_a* and *CS_b* is used as the select line for the multiplexer M2. In this case, the micro-operation $m2Out \Leftarrow m1Out$ executes whenever the value of at least one of *CS_a* and *CS_b* is one. Therefore, for this micro-operation, we can have three different CAPs, $\langle 0,1,X,X \rangle$, $\langle 1,0,X,X \rangle$, $\langle 1,1,X,X \rangle$. Therefore, f_{mc} is a non-functional relation in this example. It may be noted that even if a micro-operation μ is activated by more than one CAPs, only one of those CAPs becomes true for a given CAP from the controller. For a

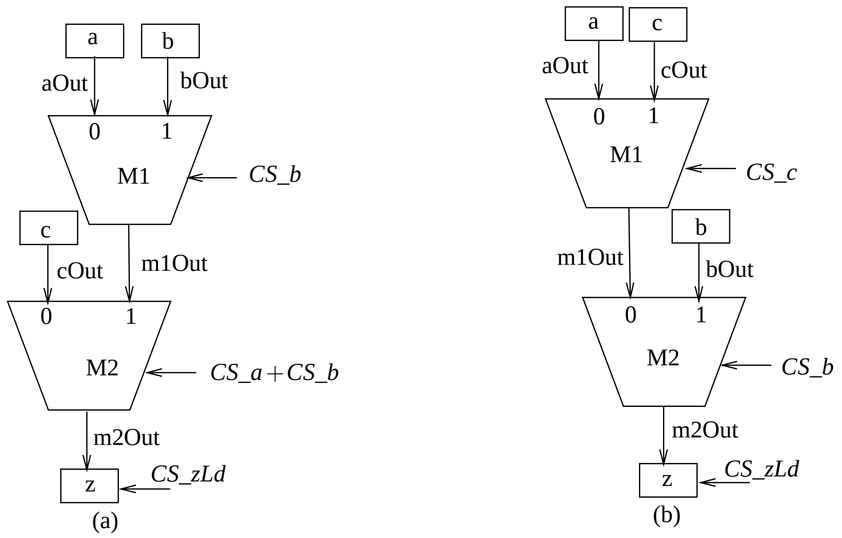


Figure 4.12: Multiplexer network restructuring to eliminate glitchy control signals: An example

given value of *CS_a* and *CS_b* in a CAP from the controller, for example, at most one of these $\langle 0,1,X,X\rangle$, $\langle 1,0,X,X\rangle$, $\langle 1,1,X,X\rangle$ is true.

Since some of the control signals are removed from the select lines of the multiplexer network, the functionality of the control signals have changed for the multiplexer network. So, the controller may have to generate different CAPs for these two networks to execute the same RT-operation in them. Also, the set of micro-operations of the transformed network is different from the original one. In this case, our rewriting method succeeds in constructing the same RT-operations from two different CAPs over two different sets of micro-operations. The details of the working of our rewriting method on the example given in figure 4.12 are omitted here for brevity.

4.7.4 Clocking of control signals

Multiplexer restructuring strategies do not always work to reduce the effect of glitches on control signals; in such cases, clocking control signals can be used to suppress glitches on control signals (Raghunathan et al., 1999).

Let us consider the design in figure 4.13(a). Let assume that both the possible control signal *CS_b* is glitchy. Since there is only one multiplexer in this circuit,

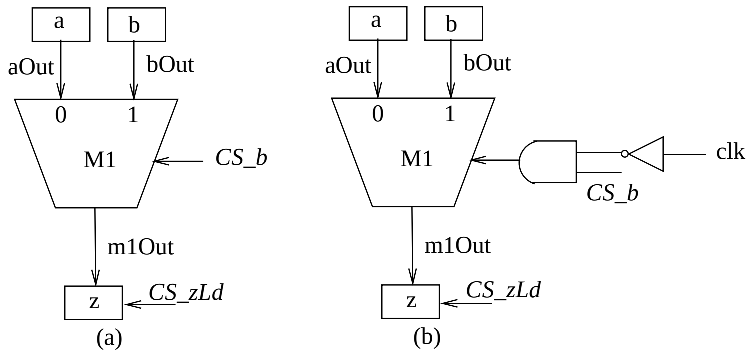


Figure 4.13: Clocking control signal to kill glitch

multiplexer restructuring transformations cannot be applied here. Let us assume that the design is implemented using rising-edge-triggered FF's and a single-phase clock with a duty cycle of 50%. Let assume that both the possible control signal *CS_b* is glitchy. In that case, multiplexer restructuring transformations cannot be applied here. Figure 4.13(b) shows the modified circuit after clocking the control signal to the multiplexer. This ensures that for the first half of the clock period, when the clock is high, the output of the AND gate is forced to zero in spite of the glitches on its other input.

This modification does not make any change in the datapath as well as in the controller. Therefore, the set of micro-operations and their corresponding CAP of the transformed RTL would be the same as that of the original RTL. The controller also generates the same CAP to execute certain RT-operation(s) in both the datapaths. Therefore, rewriting method works identically for both the behaviours.

4.7.5 Glitch reduction using delays

Let us consider the circuit in figure 4.14(a). Adders generate glitches even when the inputs are glitch free. Therefore, the data inputs to the multiplexer have glitches which propagate through the multiplexer and then through the third adder, causing significant power dissipation. The propagation of glitches can be minimized by adding delay elements (buffers or inverters) in the select line *CS_m* of the multiplexer as shown in figure 4.14(b) .

Adding a delay element or a clock to a control signal does not result in any struc-

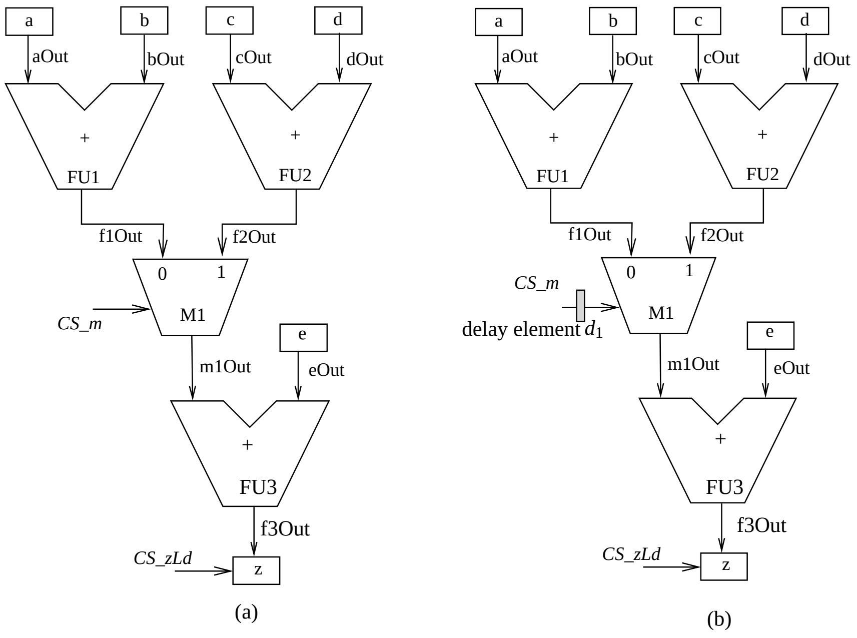


Figure 4.14: Insertion of delay element in signal line

tural changes in datapath. Also, it does not effect in the functionality of the controller. Therefore, the rewriting method works identically for both the behaviours.

Similarly, delay elements may be inserted in the data lines also. In figure 4.15, for example, the delay elements are inserted at the data line from *b* to multiplexer M1 and also at the data line from *a* to multiplexer M2. It may be noted that the multiplexer network has also restructured in this example.

When the delay elements are inserted in the datapath, number of micro-operations are increased in the datapath. Let us consider the micro-operation $m1Out \Leftarrow b$ of the datapath in figure 4.15(a). A delay element has been inserted in this data line as shown in figure 4.15(b). As a result, the micro-operation $m1Out \Leftarrow b$ is now replaced by two micro-operations $d1Out \Leftarrow d$ and $m1Out \Leftarrow d1out$. The CAPs generated by the controller to the datapath remain the original one. Since the set of micro-operations of the transformed datapath are not same as the original one, the rewriting sequence would be different in them for the same RT-operation.

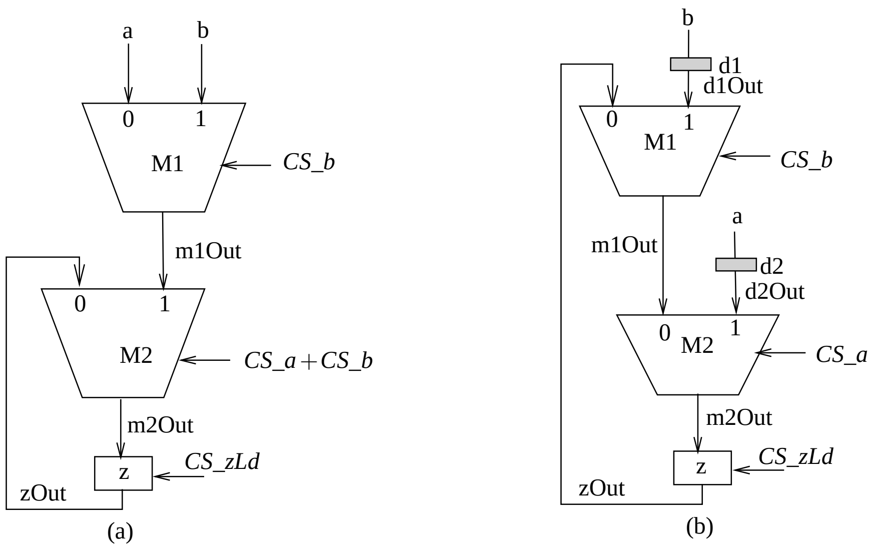


Figure 4.15: Insertion of delay element in data inputs

4.8 Experimental results

The verification method described in this chapter has been implemented in C and integrated with an existing high-level synthesis tool SAST (Karfa et al., 2005). It has been run on a 2.0 GHz Intel® Core™2 Duo machine with 2GB RAM on the outputs generated by SAST for eleven HLS benchmarks (Panda and Dutt, 1995). Some of the benchmarks such as, differential equation solver (DIFFEQ), elliptic wave filter (EWF), IIR filter (IIR_FIL) and discrete cosine transformation (DCT), are *data intensive*, some are *control intensive* such as, greatest common divisor (GCD), traffic light controller (TLC), $(a*b)$ modulo n (MODN) and barcode reader (BARCODE), whereas some are both *data and control intensive* such as, IEEE floating point unit (IEEE754), least recently used cache controller (LRU) and differential heat release computation (DHRC) (Panda and Dutt, 1995). The number of basic blocks, branching blocks, three-address operations and variables for each benchmark are tabulated in table 4.4. Before presenting the details of the experiments, a brief introduction to SAST is in order. The datapath produced by SAST may be viewed as a set of *architectural blocks* (A-blocks) connected by a set of global buses. Each A-block has a local functional unit (FU), local storage and local buses. The datapath is characterized by a three tuple of positive numbers $\langle a,b,l\rangle$, where a gives the number of A-blocks, b gives the number of global buses interconnecting the A-blocks and l gives the number

Benchmark	*#BBs*	*#branching blocks*	*#operations*	*#variables*
DIFFEQ	4	1	19	12
EWF	1	0	53	37
DCT	1	0	58	53
GCD	7	5	9	5
TLC	17	6	28	13
MODN	7	4	12	6
IIR_FIL	3	0	27	20
BARCODE	28	25	52	17
IEEE754	40	28	74	16
LRU	22	19	49	19
DHRC	7	14	131	72

Table 4.4: Characteristics of the HLS benchmarks used in our experiment

of access links or access width connecting an A-block to the global buses. SAST takes these architectural parameters along with the high-level behaviour as inputs. The tool produces different schedule of operations, different binding of variables and operators and hence, different datapath interconnection and controller from a given high-level behaviour for different architectural parameters. The design synthesized by the SAST tool under the above mentioned architectural parameters are well suited to test this verifier as the designed datapaths have complex interconnections and complex data transfers.

In our first experiment, we assume that all operations in the benchmark examples are single cycle. Two different sets of architectural parameters comprising the number of A-blocks, the number of global buses and the number of access links are considered for each benchmark depending on the size of the benchmark. The results for all the HLS benchmarks are shown in table 4.5. The number of registers, functional units, interconnection wires and switches (which is the only interconnection component here), the number of micro-operations possible in the datapath, the number of states in the controller FSM, the number of control signals used to control the micro-operations, the average time of construction of the FSMD from the datapath and the controller and the average time of verification by equivalence checking for each benchmark program for both architectural parameters are shown in columns 3-11 (under the designation “*correct design*") of this table. It may be noted that the number of control signals for each benchmark is less than the number of micro-operations in the datapath because SAST optimizes the number of control signals required to control the micro-operations in the datapath and some of the micro-operations depend on

		Correct Design									*Erroneous Design*				
	Arch.	*#regs*	*#FUs*	*#wires*	*#swi-*	*#micro-*	*#states*	*#ctrl*	*time (ms)*		*#bits*	*M_2 construction*		*Eq. Check.*	
Benchmarks	*Params.*				*tches*	*opns*		*sigs*	*construct.*	*verif*	*changes*	*#errors*	*#time*	*#errors*	*#time*
DIFFEQ	$\langle 3,2,1\rangle$	14	3	49	42	82	13	60	188	7	6	4	165	-	-
	$\langle 4,3,2\rangle$	12	4	54	48	88	10	67	170	7	2	0	183	1	8
EWF	$\langle 3,2,1\rangle$	17	3	63	83	135	34	109	504	3	10	6	443	-	-
	$\langle 4,3,2\rangle$	19	4	75	108	165	23	138	458	3	20	9	536	-	-
DCT	$\langle 3,2,1\rangle$	25	3	70	190	154	29	117	656	13	7	6	576	-	-
	$\langle 4,3,2\rangle$	31	4	96	130	218	23	171	597	14	5	1	554	-	-
GCD	$\langle 2,2,1\rangle$	4	2	24	15	29	8	21	141	21	2	0	126	1	20
	$\langle 2,1,1\rangle$	4	2	21	13	26	8	19	140	21	7	2	104	-	-
TLC	$\langle 2,2,1\rangle$	14	2	32	21	46	22	28	229	35	3	1	212	-	-
	$\langle 3,2,1\rangle$	16	3	45	23	51	22	37	205	31	10	4	260	-	-
MODN	$\langle 2,2,1\rangle$	15	2	43	33	71	15	46	144	18	8	5	162	-	-
	$\langle 3,2,1\rangle$	15	3	58	34	78	15	49	192	19	11	7	195	-	-
IIR_FIL	$\langle 3,2,1\rangle$	18	3	57	50	97	21	71	242	12	7	4	245	-	-
	$\langle 4,3,2\rangle$	21	4	81	74	131	16	99	231	10	5	0	252	2	12
IEEE754	$\langle 3,2,1\rangle$	52	3	112	110	235	120	147	1420	55	26	7	1306	-	-
	$\langle 4,3,2\rangle$	55	4	133	150	292	107	197	1260	60	16	4	1529	-	-
BARCODE	$\langle 3,2,1\rangle$	30	3	63	57	118	76	82	565	68	17	7	487	-	-
	$\langle 4,3,2\rangle$	34	4	80	81	157	58	116	510	60	3	0	592	0	55
LRU	$\langle 3,2,1\rangle$	30	3	82	105	190	72	140	1020	48	32	11	885	-	-
	$\langle 4,3,2\rangle$	33	4	102	112	217	69	152	980	46	11	5	805	-	-
DHRC	$\langle 3,2,1\rangle$	49	3	115	136	252	97	177	1445	67	44	11	1372	-	-
	$\langle 4,3,2\rangle$	57	4	139	148	289	95	198	1295	59	12	2	1387	-	-

Table 4.5: Results for several high-level synthesis benchmarks

			Correct Design									*Erroneous Design*		
			Datapath info.					*Controller info.*		*Time (ms)*			*M_2 construct.*	
Bench-marks	*operator type*	*Arch. Params.*	*#regs*	*#FUs*	*#wires*	*#swi-tchs*	*#micro-opns*	*#states*	*#ctrl sigs*	*FSMD construction*	*equiv. check*	*#bits changes*	*#errors*	*#time*
DIFFEQ	MULTICYCLE	$\langle 3,2,1\rangle$	14	3	52	51	91	13	69	211	8	3	1	200
		$\langle 4,3,2\rangle$	15	4	60	57	99	11	75	194	8	4	3	214
	PIPELINED	$\langle 3,2,1\rangle$	11	3	44	42	75	12	57	184	7	1	1	205
		$\langle 4,3,2\rangle$	13	4	58	54	99	10	75	178	7	2	2	195
DCT	MULTICYCLE	$\langle 3,2,1\rangle$	30	3	77	92	166	42	124	723	15	6	4	730
		$\langle 4,3,2\rangle$	28	4	93	126	209	27	196	726	14	3	2	722
	PIPELINED	$\langle 3,2,1\rangle$	30	3	76	91	165	36	123	629	14	4	3	642
		$\langle 4,3,2\rangle$	30	4	91	115	199	26	154	580	13	2	2	567

Table 4.6: Results for multicycle and pipelined datapath for two high-level synthesis benchmarks

more than one control signal. Our method can successfully find the RT-operations for this case. Furthermore, the number of RT-operations in each benchmark (in column 4 in table 4.4) is much higher than the number of states (in column 8 of table 4.5) in the controller FSMs. It indicates that more than one RT-operation are executed in the datapath for a control assertion pattern. Again, our method successfully finds all the RT-operations from a given control assertion pattern. The FSMD construction time is seen to increase linearly with the design size. For instances, the datapath of the GCD example for architectural parameter $\langle 2,2,1 \rangle$ consists of 14 registers, 2 FUs, 24 interconnection wires, 15 switches and 29 micro-operations; the corresponding figures for IEEE754 example for architectural parameters $\langle 3,2,1 \rangle$ are 52, 3, 112, 110 and 235. The controller for the GCD example has 8 states and 21 control signals. The corresponding figures for the IEEE754 example are 120 and 147. The construction time of the FSMD of IEEE754 is only ten times higher than that of GCD. The average FSMD construction time is higher than that of verification time. The construction time, however, is not very high and is less than two seconds in all the cases.

In our second experiment, we consider two data intensive benchmarks, i.e., DIFFEQ and DCT examples, which consist of 6 and 16 multiplications, respectively. These behaviours have been synthesized in SAST with two different architectural parameters. In addition to that, the same observations have been repeated twice; in the first step, all the multiplication operations of the behaviour are taken as two cycle ones; in the second step, these are taken as two stage pipelined. The same set of observations as in the first part of table 4.5 are carried out and recorded in table 4.6 for this experiment. Our rewriting method successfully constructed the RT-operations from the control assertion patterns for all these cases. The FSMD construction time and the verification time are not particularly high here either.

In the third experiment, we consider erroneous designs. For example, in the RTL design of IEEE754 floating point example generated by SAST with architectural parameters $\langle 3,2,1 \rangle$, we modify the 70^{th} state of the FSM. The correct RT-operations in this state are $var6 \Leftarrow exponent1 + exponent2$ and $mmult \Leftarrow mantissa1 \times mantissa2$ and the corresponding control assertion pattern is `0x1180400000020000402008000800118004080`. Now, we inject different faults in the RTL design by manually changing some bits of the control assertion pattern and check whether our method can successfully find the corresponding bugs. The set of micro-operations in this state is rendered inadequate by changing the 19^{th} hex digit of the assertion pattern from 2 to 0 (i.e., `0x118040000002000040000800080011800408 0`). As a result, the micro-operation $alu1Lin \Leftarrow exponent1Out$ is not realized in this state. The *findRewriteSeq* function finds this inconsistency as it fails to find the replacement of $alu1Lin$ during rewriting. The function reports “inadequate set of micro-operations" and returns the partially computed micro-operation sequence from the destination register $var6$ to $alu1Lin$. Similarly, data conflict is next introduced in this state by changing the assertion pattern to `0x1180400000220000400008000800118004080`. As a result, the data from both registers $r223$ and $exponent1$ try to pass to $alu1Lin$. The function *findRewriteSeq* finds two replacements, i.e., $aluLin \Leftarrow r223Out$ and $aluLin \Leftarrow exponent1Out$ for $alu1Lin$ and then reports “data conflict". In the same way, other faults are also injected and successfully found by our FSMD construction method. In the next step, we set the control assertion pattern to `0x1180400000020000402000800800118004004`. As a result, the RT-operation $mantissa \Leftarrow mantissa1 \times mantissa2$, instead of the original RT-operation $mmult \Leftarrow mantissa1 \times mantissa2$, is executed in the datapath. The *RTLV-0* constructs the RT-operation $mantissa \Leftarrow mantissa1 \times mantissa2$. Subsequently, the equivalence checking step finds this mismatch of RT-operations in the FSMD transition.

We further introduce faults by randomly changing control bits from 1 to 0 or vice versa in the controller of all the benchmark examples. The number of control bits changed, the number of errors detected during FSMD construction and equivalence checking and their respective time for this experiment are tabulated in columns 12-16 (under the designation *“erroneous design"*) of the table 4.5. It may be noted that the equivalence checking is not needed if some errors are detected during FSMD construction. In most of the cases, errors are found correctly during construction of FSMD. In some cases, incorrect or redundant RT-operations are constructed but they are found subsequently during equivalence checking. Random modification of the control bits, in some cases, resulted in (benign) partial data flows in the datapath which did not realize any RT-operation. For such modified designs, no errors were reported because the original behaviour is still realized. Our rewriting method does not reveal such partial dataflows without compromising the correctness of the method. Furthermore, for pipelined and multicycle operations, we have modified the control bits so that they are performed inconsistently. All these errors have also been detected. The number of control bits changed, the number of errors identified and the corresponding verification time are given in columns 13-15 of table 4.6. It may be noted that the execution times of the method for the erroneous design are comparable with those for the correct design. In our next part of this experiment, we introduce faults in the datapath by altering the connections of some interconnection switches without changing their control assertions (the controller circuit stays unaltered). These errors are also successfully identified by our verifier; the time taken is comparable with other experiments.

4.9 Conclusion

This chapter presents a verification method of the RTL generation phase of high-level synthesis. The verification task is performed in two steps. In the first step, a novel and formally proven *rewriting method* is presented for finding the RT-operations performed in the datapath by a given control assertion pattern. Several inconsistencies in both the datapath and the controller may be revealed during construction of the FSMD. Unlike many other reported techniques, this work provides a completely automated verification procedure of pipelined, multicycle and chained datapaths produced through high-level synthesis. Its correctness and complexity analysis are also given. In the second step, a state based equivalence checking methodology is used to verify the correctness of the controller behaviour. We next apply our FSMD construction mechanism to verify several RTL low power transformations. Specifically, we construct FSMDs from both the initial and the transformed RTLs and then apply our FSMD based equivalence checker. Experimental results on several HLS benchmarks demonstrate the effectiveness of our method.

Chapter 5

Verification of Loop and Arithmetic Transformations of Array-Intensive Behaviours

5.1 Introduction

In course of synthesizing the final implementation from the initial specification of an algorithm, a set of transformations may be carried out on the input behaviour targeting the best performance, energy and/or area on a given platform. In particular, loop transformations along with algebraic and arithmetic transformations are applied extensively in the domain of multimedia and signal processing applications. These transformations can be automatic, semi-automatic or manual. We have discussed in section 2.3.1 that loop transformation techniques like loop fusion, loop tiling, loop shifting, loop unrolling, loop spliting, etc., are used quite often in the multimedia and signal processing domain. We have also discussed in section 2.3.2 that arithmetic transformations based on algebraic properties of the operator like associativity, commutativity and distributivity, arithmetic expression simplification, constant folding, common subexpression elemination, copy propagation, renaming and operator strength reduction, etc., are also applied during synthesis. Importantly, loop transformation and arithmetic transformation techniques are applied dynamically since application of one may create the scope of application of the other. In all cases, it is crucial to know that the transformed program preserves the behavior of the original.

In this chapter, we develop an equivalence checking method which is capable of handling all kinds of loop transformations and a wide range of arithmetic transformations applied on array intensive behaviours. We model both the input behaviour and the transformed behaviour as array data dependence graph (ADDG) models (Shashidhar, 2008). As discussed in subsection 2.3.3, an ADDG based equivalence checking method for loop and data-flow transformations was proposed in (Shashidhar, 2008; Shashidhar et al., 2005a). This method works for most of the loop transformations and for associative and commutative transformations. Their method, however, fails if the transformed behaviour is obtained from the original behaviour by application of arithmetic transformations such as, distributive transformations, arithmetic expression simplification, common sub-expression elimination, constant folding, substitution of multiplications with constants by addition, etc, along with loop transformations. As discussed in (Shashidhar, 2008; Shashidhar et al., 2005a), to hanlde further arithmetic transformations, one has to evolve appropriate rule for each individual transformation and apply that rule to transform the ADDGs accordingly, before matching their slices. This suggested direction of extension of their mathod, , however, cannot be extended to handle many of above mentioned arithmetic transformations. It is because of the fact that the defintion of equivalence of slices developed in their work necessiates that the number of paths must be the same in two equivalent slices. This assumption, however, may not hold in several cases of arithmetic transformations. In this work, we define a slice based equivalence of ADDGs to alleviate the restrictions of the method proposed in (Shashidhar, 2008; Shashidhar et al., 2005a). Specifically, the contributions of the present chapter are: 1. modification of definition of the characteristic formula of a slice of ADDGs, 2. redefining the equivalence of ADDGs based on slice level characterization rather than path based one alleviating, in the process, a shortcoming of the latter in handling equivalent slices with unequal number of paths, 3. incorporating normalization of arithmetic expressions (Sarkar and De Sarkar, 1989) and some additional simplification rules for normalized expressions for handling several arithmetic transformations applied along with loop transformations, 4. automating method for checking equivalence of ADDGs and 5. providing the correctness and complexity of the proposed method.

The rest of the chapter is organized as follows. The ADDG model is introduced in section 5.2, The notions of slice and its characteristic formula are defined in section

5.3. The equivalence checking method of ADDGs is given in section 5.4. The method is explained with an example in section 5.5. Correctness and complexity issues are dealt with in section 5.6. Several kinds of errors and corresponding diagonistic inferences are discussed in section 5.7. Some experimental results are given in section 5.8. Finally, the chapter is concluded in section 5.9.

5.2 Array data dependence graphs

A sequential program consists of a set of statements. The right-hand-side (rhs) expression of any assignment statement is treated as an arithmetic expression represented as a function f called the operator in the statement. The array in the left hand side (lhs) depends on the operator. The operator, in turn, depends on the arrays occurring in the right hand side (rhs) expression. In the case of a copy statement, where one just copies the contents of one array to another array, we consider the operator to be the identity function I. Each individual array and each operator in a statement of the behaviour form the vertex set and the dependencies discussed above form the edge set in the ADDG. Hence we have the following definition.

Definition 14 (Array Data Dependence Graph (ADDG):) *The ADDG of a sequential behaviour is a directed graph $G = (V,E)$, where the vertex set V is the union of the set A of array nodes and the set F of operator nodes and the edge set $E = \{\langle a, f\rangle \mid a \in A, f \in F\} \cup \{\langle f,a\rangle \mid f \in F, a \in A\}$. Edges of the form $\langle a, f\rangle$ are write edges; they capture the dependence of the lhs array node on the operator corresponding to the rhs expression. The edges of the form $\langle f,a\rangle$ are read edges; they capture the dependence of the rhs operator on the (rhs) operand arrays. An assignment statement S of the form $l[\vec{e}_l] = f(r_1[\vec{e}_1],\ \ldots,\ r_k[\vec{e}_k])$, where $\vec{e}_1,\ \ldots,\ \vec{e}_k$ and $\vec{e}_l$ are the vectors of index expressions of the arrays $r_1,\ \ldots,\ r_k$ and l, appears as a subgraph G_S of G, where $G_S = \langle V_S,\ E_S\rangle$, $V_S = A_S \cup F_S$, $A_S = \{l, r_1,\ \ldots,\ r_k\} \subseteq A$, $F_S = \{f\} \subseteq F$ and $E_S = \{\langle l, f\rangle\} \cup \{\langle f, r_i\rangle,\ 1 \le i \le k\rangle\} \subseteq E$. The write edge $\langle l, f\rangle$ is associated with the statement name S. If the operator associated with an operator node f has an arity k, then there will be k read edges $\langle f,a_1\rangle,\ \ldots,\ \langle f,a_k\rangle$. The operator f applies over k arguments which are elements of the arrays $a_1,\ldots,a_k$, not all distinct.*

A sequential program can be represented as an ADDG under the following restrictions: *single-assignment, affine indices and bounds, static control-flow and uniform recurrences* (Shashidhar, 2008). Let us now describe each of the restrictions.

Single-assignment: A behaviour is said to be in single assignment form if any memory location is written at most once in the entire behaviour. In other words, each individual variable and each location of an array can be defined at most once and cannot be redefined.

Affine indices and bounds: The array index expressions and the loop bounds must be affine in nature. It means that all arithmetic expressions in the indices of array variables and in the bounds of loops must be affine function in the iteration variables of the enclosing loops. An affine transformation, or affine map, or affinity, between two vector spaces V_1 and V_2 (strictly speaking, two affine spaces) consists of a linear transformation followed by a translation such as, $x \leftarrow Ax + b$, where A is a matrix over the underlying scalar of the vector space and b is a vector in V_1 [1]. For the nested loop behaviours, the ordered tuples of iteration variables of the enclosing loops constitute a vector space.

Static control-flow: The control-flow of a behaviour is static if the execution of the program can be exactly determined in compile time.

Uniform recurrences: A recurrence of the form $M(\vec{i}) = \vec{i} + \vec{c}$, where $\vec{i}$ is a vector of integer variables and $\vec{c}$ is a vector of constant integers, is called uniform recurrence (Karp et al., 1967).

In a single-assignment form program, any variable other than the loop indices, can be replaced with an array variable (Shashidhar, 2008). Therefore, without loss of generality, it is assumed that all the statements in the behaviours involve only array variables. In fact, the application domain that we are considering also consists of array intensive behaviours.

[1] In the finite-dimensional case, each affine transformation is given by a matrix A and a vector b, satisfying certain properties described below.

Geometrically, an affine transformation in Euclidean space is one that preserves

(i) The collinearity relation between points; i.e., three points which lie on a line continue to be collinear after the transformation. (ii) Ratios of distances along a line; i.e., for distinct collinear points p1, p2, p3, the ratio $|p2 - p1| \ / \ |p3 - p2|$ is preserved.

As the program is considered to be in single assignment form, an element of an array cannot be defined through two different operators. On the other hand, if $a^{(i)}$ and $a^{(j)}$ be the sets of elements of the array a defined respectively through the operator f_i and f_j, $1 \leq i, j \leq l \wedge i \neq j$, then $a^{(i)}$ and $a^{(j)}$ must be two disjoint sets of elements. So, if the elements of an array a are defined through l operators, $f_1, \ldots, f_l$, then the array node a will be associated with l write edges of the form $\langle a, f_1 \rangle, \ldots, \langle a, f_l \rangle$ in the ADDG. Also, even if a program has more than one statement with reference to the same array a in the lhs, there will be only one array node in the ADDG for the array a.

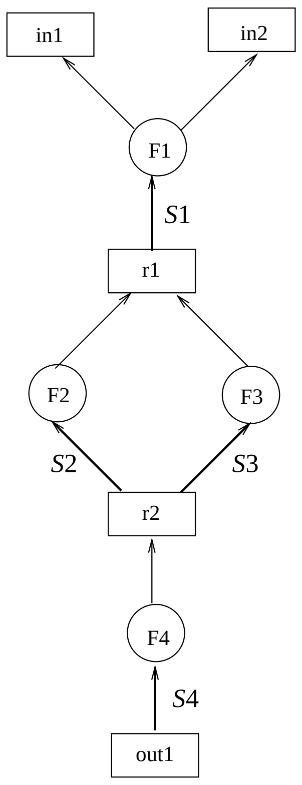


Figure 5.1: The ADDG of the sequential behaviour given in example 10

Example 10 Let us now introduce one such behaviour through the following example. Rest of the formalism will be explained through this behaviour.

The following nested loop sequential behaviour first computes the elements of the array $r1$ from values of two input arrays namely, $in1$ and $in2$. In the next loop, the program computes the values of array $r2$ and finally produces the output array $out1$ from $r2$.

```
**for** ($i = 1;\ i \leq M;\ i+ = 1$) **do**
    **for** ($j = 4;\ j \leq N;\ j+ = 1$) **do**
        $r1[i+1][j-3] = F1(in1[i][j],\ in2[i][j]);$    // S1
    **end for**
**end for**
**for** ($l = 3;\ l \leq M;\ l+ = 1$) **do**
    **for** ($m = 3;\ m \leq N-1;\ m+ = 1$) **do**
        **if** ($l+m\ \leq\ 7$) **then**
            $r2[l][m] = F2(r1[l-1][m-2]);$    // S2
        **end if**
        **if** ($l+m\ \geq\ 8$) **then**
            $r2[l][m] = F3(r1[l][N-3]);$    // S3
        **end if**
    **end for**
**end for**
**for** ($l = 3;\ l \leq M;\ l+ = 1$) **do**
    **for** ($m = 3;\ m \leq N-1;\ m+ = 1$) **do**
        $out1[l][m] = F4(r2[l][m]);$    // S4
    **end for**

**end for**
```

The ADDG of this sequential behaviour is given in figure 5.1. The arrays $in1, in2$, $r1$, $r2$ and $out1$ account for the array nodes represented as rectangles and the operators (functions) $F1, F2, F3$ and $F4$ account for the operator nodes represented as circles in the ADDG. The dependencies among the nodes are represented as directed edges as shown in the figure. In the statement $S1$, for example, the elements of the array $r1$ is defined in terms of the elements of the arrays $in1$ and $in2$ using the function $F1$. Accordingly, there are two read edges $\langle F1, in1\rangle$, $\langle F1, in2\rangle$ and one write edge $\langle r1, F1\rangle$ in the ADDG. The statement name $S1$ is associated with the write edge $\langle r1, F1\rangle$. The other edges in the ADDG can also be obtained in the same way from the other statements of the behaviour. □

```
$for(i_1 = L_1; i_1 \leq H_1; i_1+ = r_1)$
  $for(i_2 = L_2; i_2 \leq H_2; i_2+ = r_2)$
     $\vdots$
       $for(i_x = L_x; i_x \leq H_x; i_x+ = r_x)$
           $if(C_D)\ then$
           $d[e_1]\ldots[e_k] = f(u_1[e'_{11}]\ldots[e'_{1l_1}],\ \ldots,\ u_m[e'_{m1}]\ldots[e'_{ml_m}]);$   $//statement\ S$
```

Figure 5.2: A generalized x-nested loop structure

5.2.1 Representation of data dependencies of the behaviour

The following information is extracted from each statement S of the behaviour. Let us consider the generalized x-nested loop structure in figure 5.2 for this purpose. Each index i_k has a lower limit L_k and a higher limit H_k and a step constant (increment) r_k. The parameters L_k, H_k and r_k are all integers. The statement S executes under the condition C_D within the loop body. The condition C_D is a logical expression involving relational expressions over the loop indices. The array being defined is d and the arrays read are $u_1, \ldots, u_m$. All the index expressions $e_1, \ldots, e_k$ of d and the index expressions $e'_{11}, \ldots, e'_{1l_1}, \ldots, e'_{m1}, \ldots, e'_{ml_m}$ of the corresponding arrays $u_1, \ldots, u_m$ in the statement S are affine arithmetic expressions over the loop indices. There may be other statements under the same condition or under other conditions within the loop body. They are not considered here for simplicity.

Definition 15 (Iteration domain of the statement S (I_S)) *For each statement S within a generalized loop structure with nesting depth x, the iteration domain I_S of the statement is a subset of $\mathbb{Z}^x$ defined as*

$$I_S = \{[i_1, i_2, \ldots, i_x] \mid \bigwedge_{k=1}^{x} (L_k \le i_k \le H_k \wedge C_D \wedge \exists \alpha_k \in \mathbb{N}(i_k = \alpha_k r_k + L_k))\}$$

where $i_k, L_k, H_k, r_k, 1 \le k \le x$, are integers [2].

Example 11 Let us consider the statement $S1$ of the example 10. For this statement,

$$I_{S1} = \{[i,\ j] \mid 1 \le i \le M \wedge 4 \le j \le N \wedge (\exists \alpha_1, \alpha_2 \in \mathbb{N} \mid i = \alpha_1 + 1 \wedge j = \alpha_2 + 4)\}$$

Similarly, for the statement $S2$ of the same example, the iteration domain is

$I_{S2} = \{[l,\ m] \mid 3 \le l \le M \wedge 3 \le m \le N - 1 \wedge l + m \le 7 \wedge (\exists \alpha_1, \alpha_2 \in \mathbb{N} \mid l = \alpha_1 + 3 \wedge m = \alpha_2 + 3)\}$ □

[2] Alternative definition:

$$I_S = \{[i_1, i_2, \ldots, i_x] \mid \bigwedge_{k=1}^{x} (L_k \le i_k \le H_k \wedge C_D \wedge (i_k - L_k)\%r_k = 0)\}$$

This definition removes the extra variable α.

Each point $[i_1, i_2, \ldots, i_x]$ in I_S denotes exactly one execution of the statement S. Each statement of a program defines the elements of an array (occurring in the lhs) using the elements of a set of operand arrays (appearing in the rhs). Therefore, for each statement, we may speak of a *definition domain* and a *definition mapping* of the lhs array in the statement. The former comprises the domain from which the lhs array indices assume values and the latter provides the mapping between the elements of the iteration domain of the statement to the definition domain of the lhs array. Similarly, for each rhs array of each statement, we have an *operand domain* and an *operand mapping* of the rhs array in the statement. The former comprises the domain from which the indices of rhs array of the statement assume values and the latter provides the mapping between the elements of the iteration domain of the statement to the operand domain of the rhs array. There are one operand domain and one operand mapping for each of the arrays in the rhs of the statement. The following definitions are in order.

Definition 16 (Definition mapping ($_SM_d^{(d)}$)) *The definition mapping describes the association between the elements of the iteration domain of a statement and the elements of its lhs array d.*

$$_SM_d^{(d)} = I_S \rightarrow \mathbb{Z}^k \text{ s.t. } \forall \vec{v} \in I_S,\ \vec{v} \mapsto [e_1(\vec{v}),\ \ldots,\ e_k(\vec{v})] \in \mathbb{Z}^k.$$

The image of the function $_SM_d^{(d)}$ is called the *definition domain* of the lhs array d, defined as $_SD_d$; So, $_SD_d = {}_SM_d^{(d)}(I_S)$. Due to single assignment form, each element of the iteration domain of a statement defines exactly one element of the lhs array of the statement. Therefore, the mapping between the iteration domain of the statement and image of $_SM_d^{(d)}$ is injective (one-one). Hence, $(_SM_d^{(d)})^{-1}$ exists.

In a similar way, the operand mapping for each operand array of a statement can be defined as follows.

Definition 17 (Operand mapping ($_SM_{u_n}^{(u)}$)) *The operand mapping describes the association between the elements of the iteration domain of a statement and the elements of the rhs array u_n*

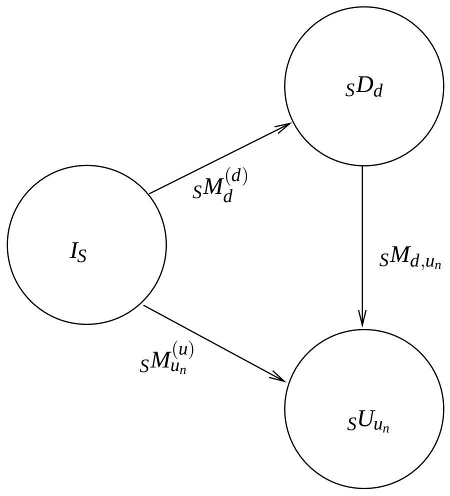
$$_SM_{u_n}^{(u)} = I_S \rightarrow \mathbb{Z}^{l_n} \text{ s.t. } \forall \vec{v} \in I_S,\ \vec{v} \mapsto [e_{n1}(\vec{v}),\ \ldots,\ e_{nl_n}(\vec{v})] \in \mathbb{Z}^{l_n}.$$

The image ${}_SM_{u_n}^{(u)}(I_S)$ is the *operand domain* of the rhs array u_n in the statement S, denoted as ${}_SU_{u_n}$. One element of the operand array u_n may be used to define more than one element of the array d. It means that more than one element of the iteration domain may be mapped to one element of the operand domain. Hence, ${}_SM_{u_n}^{(u)}$ may not be injective.

Definition 18 (Dependence mapping (${}_SM_{d,u_n}$)) *It describes the association of the index expression of the lhs array d which is defined through S to the index expression of the operand array $u_n, 1 \leq n \leq m$, i.e., one of the rhs arrays of S.*

$$\begin{aligned} {}_SM_{d,u_n} = \; & \{[e_1,\ldots,e_k] \rightarrow [e'_1,\ldots,e'_{l_n}] \mid ([e_1,\ldots,e_k] \in {}_SD_d \;\wedge\; [e'_1,\ldots,\, e'_{l_n}] \in {}_SU_{u_n} \;\wedge \\ & \exists \vec{v} \in I_S \mid ([e_1,\ldots,e_k] = {}_SM_d^{(d)}(\vec{v}) \;\wedge\; [e'_1,\ldots,e'_{l_n}] = {}_SM_{u_n}^{(u)}(\vec{v})))\} \end{aligned}$$

The defined array d is k-dimensional and $e_1,\ldots,e_k$ are its index expressions over the loop indices $\vec{v}$; the array u_n is an l_n-dimensional array, and $e'_1,\ldots,e'_{l_n}$ are its index expressions over the indices $\vec{v}$.



$${}_SM_{d,u_n} \;=\; ({}_SM_d^{(d)})^{-1} \diamond {}_SM_{u_n}^{(u)}$$

Figure 5.3: Computation of dependence mapping

The dependence mapping ${}_SM_{d,u_n}$ can be obtained by

$${}_SM_{d,u_n} \;=\; ({}_SM_d^{(d)})^{-1} \diamond {}_SM_{u_n}^{(u)}$$

Specifically, ${}_{S}M_{d,u_n}({}_{S}D_d) = {}_{S}M^{(u)}_{u_n}(({}_{S}M^{(d)}_{d})^{-1}({}_{S}D_d)) = {}_{S}M^{(u)}_{u_n}(I_S) = {}_{S}U_{u_n}$. Figure 5.3 illustrates the fact.

There would be one such mapping from the defined array to each of the operand arrays in the rhs of S. The mappings ${}_{S}M_{d,u_n}$, $1 \le n \le m$, will be associated with the corresponding read edge $\langle f, u_n \rangle$ in the ADDG.

Example 12 Let us consider the statement $S1$ of the example 10 again. For this statement,

$${}_{S1}M^{(d)}_{r1} = \{[i,j] \to [i+1, j-3] \mid 1 \le i \le M \wedge 4 \le j \le N\}$$

$${}_{S1}D_{r1} = \{[i+1, j-3] \mid 1 \le i \le M \wedge 4 \le j \le N\}$$

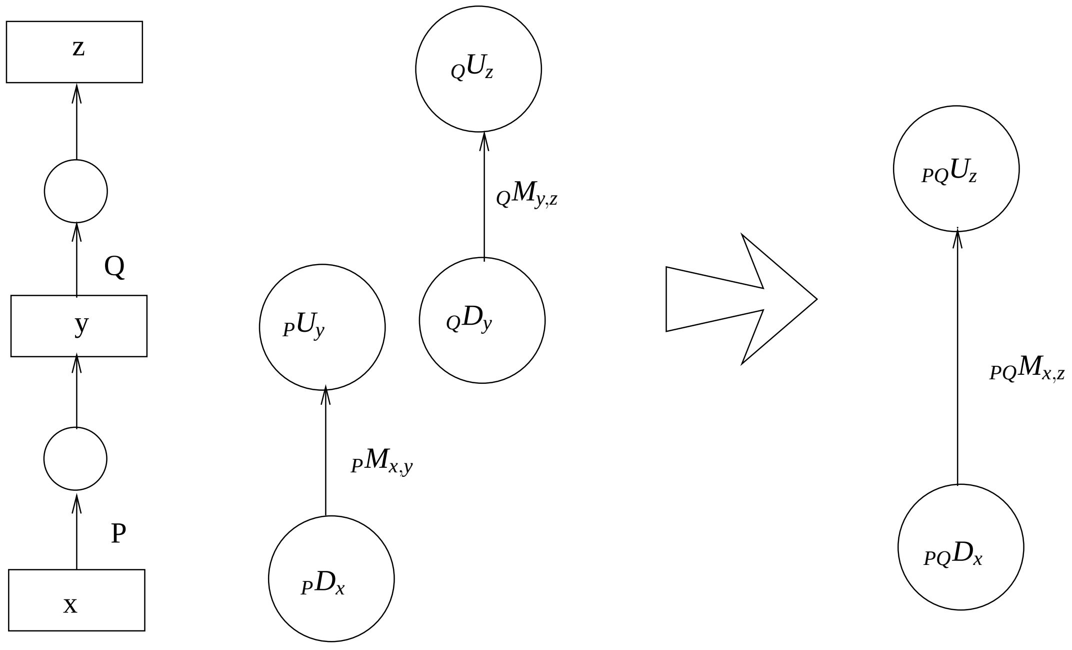
$${}_{S1}M^{(u)}_{in1} = \{[i,j] \to [i,j] \mid [i,j] \in 1 \le i \le M \wedge 4 \le j \le N\}$$

$${}_{S1}U_{in1} = \{[i,j] \mid 1 \le i \le M \wedge 4 \le j \le N\}$$

$$\begin{aligned} {}_{S1}M_{r1,in1} &= ({}_{S1}M^{(d)}_{r1})^{-1} \diamond {}_{S1}M^{(u)}_{in1} \\ &= \{[i,j] \to [i-1, j+3] \diamond [i-1, j+3] \to [i-1, j+3]\} \mid [i,j] \in {}_{S1}D_{r1}\} \\ &= \{[i,j] \to [i-1, j+3] \mid [i,j] \in {}_{S1}D_{r1}\} \qquad \square \end{aligned}$$

5.2.2 Transitive dependence

Data dependence exists between two statements P and Q if Q defines the values of one array and P subsequently reads the same values from that array to define another array. Figure 5.4 depicts such a situation. Let the elements of the array y be defined in terms of the elements of the array z in the statement Q and subsequently (a subset of) the elements of y defined in Q be used to define the elements of the array x in the statement P, as depicted in figure 5.4. Corresponding to the statements P and Q, we have the dependence mappings ${}_{P}M_{x,y}$ and ${}_{Q}M_{y,z}$, respectively. Therefore, the elements of the array x depends transitively on the elements of the array z. The dependence mapping between the array x and the array z, i.e., ${}_{PQ}M_{x,z}$, can be obtained from the



$$
\begin{aligned}
{}_{PQ}D_x &= {}_PM^{-1}_{x,y}({}_QM^{-1}_{y,z}({}_QU_z) \cap {}_PM_{x,y}({}_PD_x)) \\
{}_{PQ}U_z &= {}_QM_{y,z}[{}_PM_{x,y}({}_PD_x) \cap_Q M^{-1}_{y,z}({}_QU_z)]
\end{aligned}
$$

Figure 5.4: Transitive dependence

mappings ${}_PM_{x,y}$ and ${}_QM_{y,z}$ by right composition ($\diamond$) of ${}_PM_{x,y}$ and ${}_QM_{y,z}$. The following definition captures this computation.

Definition 19 (Transitive Dependence) ${}_{PQ}M_{x,z} = {}_PM_{x,y} \diamond {}_QM_{y,z} = \{[e_1,\ldots,e_{l_1}] \rightarrow [e''_1,\ldots,e''_{l_3}] \mid \exists[e'_1,\ldots,e'_{l_2}]$ `s.t.` $[e_1,\ldots,e_{l_1}] \rightarrow [e'_1,\ldots,e'_{l_2}] \in {}_PM_{x,y} \wedge [e'_1,\ldots,e'_{l_2}] \rightarrow [e''_1,\ldots,e''_{l_3}] \in {}_QM_{y,z}\}$, *where y is used in P and is defined in Q; we say that the array y satisfies "used-defined" relationship over the sequence P, Q of statements.*

The operand domain of y (${}_PU_y$) in the statement P and the definition domain of y (${}_QD_y$) in the statement Q may not be the same. This can happen because a subset of the elements of y used in P as operands may be written (defined) by a statement other than Q leading to branching of dependencies of the destination array through other paths. This situation is depicted by ${}_PU_y \supseteq {}_QD_y$. It is also possible that ${}_PU_y \subseteq {}_QD_y$ when only a subset of elements of y defined in Q is used as operands in the statement P (the remaining elements of ${}_QD_y$ may be used by other statements). Therefore, the domain ${}_{PQ}D_x$ say, of the resultant dependence mapping (i.e., ${}_{PQ}M_{x,z}$) is the subset of the domain of ${}_PM_{x,y}$ such that $\forall x \in {}_{PQ}D_x$, ${}_PM_{x,y}(x) = \{y \mid y \in range({}_PM_{x,y})$

$\cap\ domain(_Q M_{y,z})\}$. Thus, $_{PQ}D_x = \ _P M^{-1}_{x,y}(_Q M^{-1}_{y,z}(_Q U_z) \cap {_P M_{x,y}}(_P D_x))$. Similarly, the range of $_{PQ}M_{x,z}$ is $_{PQ}U_z = \ _Q M_{y,z}(_P M_{x,y}(_P D_x) \cap {_Q M^{-1}_{y,z}}(_Q U_z))$. This is illustrated in figure 5.4. The operation $\diamond$ returns empty if $_P U_y \cap {_Q D_y}$ is empty which indicates that the iteration domains of P and Q are non-overlapping.

It may be noted that the definition of transitive dependence can be extended over a sequence of statements (by associativity) and also over two sequences of statements.

Example 13 Let us consider the behaviour given in example 10 and its corresponding ADDG of figure 5.1. Let us now consider the statements $S4$ and $S2$ of the behaviour. We have

$I_{S4} = \{[l,\ m] \mid 3 \leq l \leq M \wedge 3 \leq m \leq N-1 \wedge (\exists \alpha_1, \alpha_2 \in \mathbb{Z} \mid l = \alpha_1 + 3 \wedge m = \alpha_2 + 3)\}$,

$_{S4}D_{out1} = I_{S4}$, $_{S4}U_{r2} = I_{S4}$

$_{S4}M_{out1,r2} = \{[l,m] \rightarrow [l,m] \mid [l,m] \in {_{S4}D_{out1}}\}$,

$I_{S2} = \{[l,\ m] \mid 3 \leq l \leq M \wedge 3 \leq m \leq N-1 \wedge l+m \leq 7 \wedge (\exists \alpha_1, \alpha_2 \in \mathbb{Z} \mid l = \alpha_1 + 3 \wedge m = \alpha_2 + 3)\}$,

$_{S2}D_{r2} = I_{S2}$,

$_{S2}U_{r1} = \{[l-1,\ m-2] \mid [l,m] \in I_{S2}\}$ and

$_{S2}M_{r2,r1} = \{[l,m] \rightarrow [l-1,m-2] \mid [l,m] \in {_{S2}D_{r2}}\}$.

The transitive dependence mapping $_{S4S2}M_{out1,r1}$ can be obtained from $_{S4}M_{out1,r2}$ and $_{S2}M_{r2,r1}$ by the composition operator $\diamond$ as follows:

$$\begin{aligned}
{_{S4S2}M_{out1,r1}} &= {_{S4}M_{out1,r2}} \diamond {_{S2}M_{r2,r1}} \\
&= \{[l,m] \rightarrow [l,m] \mid [l,m] \in {_{S4}D_{out1}}\} \diamond \{[l,m] \rightarrow [l-1,m-2] \mid [l,m] \in {_{S2}D_{r2}}\} \\
&= [l,m] \rightarrow [l-1,m-2] \mid [l,m] \in {_{S4}D_{out1}}\}
\end{aligned}$$

We have $_{S2}M_{r2,r1} :\ {_{S2}D_{r2}} \rightarrow {_{S2}U_{r1}}$. So, $_{S2}M^{-1}_{r2,r1}(_{S2}U_{r1})$ would be $_{S2}D_{r2}$ which is actually I_{S2}. We also have $_{S4}M_{out1,r2} :\ {_{S4}D_{out1}} \rightarrow {_{S4}U_{r2}}$. So, $_{S4}M_{out1,r2}(_{S4}D_{out1})$

would be ${}_{S4}U_{r2}$ which is actually I_{S4}. Therefore, the domain of ${}_{S4S2}M_{out1,r1}$ is ${}_{S4S2}D_{out1}$ $= {}_{S4}M^{-1}_{out1,r2}({}_{S2}M^{-1}_{r2,r1}({}_{S2}U_{r1}) \cap {}_{S4}M_{out1,r2}({}_{S4}D_{out1})) = {}_{S4}M^{-1}_{out1,r2}(I_{S2} \cap I_{S4}) = I_{S2}$. It may be noted that ${}_{S2}D_{r2}$ (which is I_{S2}) is a subset of ${}_{S4}U_{r2}$ (which is I_{S4}) in this example. (The remaining part of ${}_{S4}U_{r2}$ is defined by the statement $S3$.) The range of ${}_{S4S2}M_{out1,r1}$ is ${}_{S4S2}U_{r1} = {}_{S4S2}U_{r1} = {}_{S2}M_{r2,r1}({}_{S4}M_{out1,r2}({}_{S4}D_{out1}) \cap {}_{S2}M^{-1}_{r2,r1}({}_{S2}U_{r1}))$ $= {}_{S2}M_{r2,r1}\,(I_{S4} \cap I_{S2}) = I_{S2}$.

We also have

$$I_{S1} = \{[i,\ j] \mid 1 \le i \le M \wedge 4 \le j \le N \wedge (\exists \alpha_1, \alpha_2 \in \mathbb{Z} \mid i = \alpha_1 + 1 \wedge j = \alpha_2 + 4)\},$$

${}_{S1}D_{r1} = \{[i+1, j-3] \mid [i,j] \in I_{S1}\}$ and

$${}_{S1}M_{r1,in1} = \{[i+1, j-3] \to [i,j] \mid [i,j] \in {}_{S1}D_{r1}\}$$

The transitive dependence mapping ${}_{S4S2S1}M_{out1,in1}$ over the sequence $S4, S2, S1$ of statements can be obtained from ${}_{S4S2}M_{out1,r1}$ and ${}_{S1}M_{r1,in1}$ in the following way:

$$\begin{aligned}
{}_{S4S2S1}M_{out1,in1} &= {}_{S4S2}M_{out1,r1} \diamond {}_{S1}M_{r1,in1} \\
&= \{[l,m] \to [l-1, m-2] \mid [l,m] \in {}_{S4}D_{out1}\} \\
&\quad \diamond \{[i+1, j-3] \to [i,j] \mid [i,j] \in {}_{S1}D_{r1}\} \\
&= \{[l,m] \to [l-2, m+1] \mid [l,m] \in {}_{S4}D_{out1}\}
\end{aligned}$$

Specifically, the right composition operator $\diamond$ checks that $\forall [l,m] \in {}_{S4}D_{out1}, \exists [i,j] \in {}_{S1}D_{r1}$ such that $l-1 = i+1$, i.e., $i = l-2$. Similarly, $m-2 = j-3$, i.e., $j = m+1$.

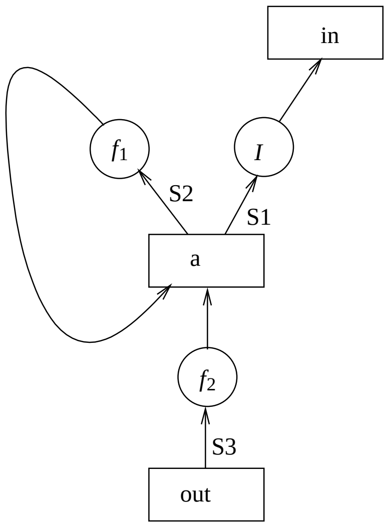
We have ${}_{S1}M_{r1,in1} : {}_{S1}D_{r1} \to {}_{S1}U_{in1}$. So, ${}_{S1}M^{-1}_{r1,in1}({}_{S1}U_{in1}) = {}_{S1}D_{r1}$. We also have ${}_{S4S2}M_{out1,r1} : {}_{S4S2}D_{out1} \to {}_{S4S2}U_{r1}$. So, ${}_{S4S2}M_{out1,r1}({}_{S4S2}D_{out1}) = {}_{S4S2}U_{r1} = I_{S2}$. So, the domain of ${}_{S4S2S1}M_{out1,in1}$ is ${}_{S4S2S1}D_{out1} = {}_{S4S2}M^{-1}_{out1,r1}({}_{S1}M^{-1}_{r1,in1}({}_{S1}U_{in1})$ $\cap\ {}_{S4S2}M_{out1,r1}({}_{S4S2}D_{out1})) = {}_{S4S2}M^{-1}_{out1,r1}({}_{S1}D_{r1} \cap I_{S2}) = I_{S2}$ (since I_{S2} is a subset of ${}_{S1}D_{r1}$). It can also be shown that the range of ${}_{S4S2S1}D_{out1}$ is I_{S2}. □

5.2.3 Recurrence in ADDG

An ADDG may contain a cycle. Since we consider programs in single-assignment form, a cycle in a ADDG does not imply a circular data dependence among the array elements. It simply means that there exists a set of statements that define arrays and these arrays depend on themselves for values assigned by the same set of statements in their earlier execution. The statements involved are then said to define a recurrence in the dependencies in the data-flow (Shashidhar, 2008). The transitive dependence mapping over a recurrence is called *across recurrence mapping*. Since we consider only uniform recurrences, it is always possible to compute this mapping for such a cycle directly without completely enumerating the recurrence. For computing across recurrence mapping in such a cycle in an ADDG, we use the technique proposed in (Shashidhar, 2008) and explained below with the help of an example.

```
S1: a[0] = in[0];
for(i=1; i<=N; i+=1)
   S2: a[i] = f1(a[i-1]);
S1: out[0] = f2(a[N]);
```

(a)



(b)

Figure 5.5: (a) A program with recurrence; (b) the ADDG of the program

Example 14 In the program in figure 5.5(a), the value of the array location $a[i],\ 1 \leq i \leq N$ depends on $a[i-1]$. This recurrence of the program results in a cycle $c = (a, f_1, a)$ in the ADDG as shown in figure 5.5(b). Let us now compute the across-recurrence transitive mapping from a to a, i.e, ${}_{S2}M_{a,a}$ and then use it to compute the transitive dependence mapping over this recurrence from *out* to a. Since, there is only one statement involved in the recurrence path, the transitive mapping ${}_{c}M_{a,a}$ is given by

$${}_{c}M_{a,a} = \{[i] \to [i-1] \mid 1 \le i \le N\}$$

The transitive closure of this mapping is given by

$$\begin{aligned} m &= (\, {}_{c}M_{a,a}\,)^{+} \\ &= \{i \to \{(\, {}_{c}M_{a,a}\,)^{k}(i)\} \mid k \ge 1 \wedge 1 \le i \le N\} \\ &= \{[i] \to [P] \mid 0 \le i \le N \wedge P = \{\alpha \mid 0 \le \alpha < i\}\}. \end{aligned}$$

Note that k will be less than equal to N for present case of recurrence. The domain and range of m are given by,

$$d = domain(m) = \{i \mid 0 < i \le N\}.$$

$$r = range(m) = \{i \mid 0 \le i < N\}.$$

The end-to-end mapping of any ADDG cycle specifies the mapping of the largest index value. In this case, therefore, it is given by $N \mapsto (\, {}_{c}M_{a,a}\,)^{N}(N) = 0$. Note that when conceived as a function, the domain and the range of the end-to-end mapping are obtained from the domain and the range of the transitive closure as

$$d' = (d - r) = \{N\} \text{ and}$$

$$r' = (r - d) = \{0\}.$$

Since the domain d' and the range r' of the end-to-end mapping are both unit sets, the across-recurrence mapping is

$${}_{S2}M'_{a,a} = (m \backslash d')/r' = \{[N] \to [0]\}.$$

Now, the transitive dependence over the recurrence from *out* to *a* can be computed as

$${}_{S3S2}M_{out,a} = {}_{S3}M_{out,a} \diamond {}_{S2}M_{a,a} = [0] \to [N] \diamond [N] \to [0]\} = \{[0] \to [0]\}. \qquad \square$$

5.2.4 Construction of the ADDG from a sequential behaviour

Let us now discuss how the ADDG of a given sequential program can be obtained. Initially, the ADDG $G = (V, E)$ is empty, i.e., $V = \emptyset$ and $E = \emptyset$. We read the behaviour sequentially. Corresponding to a statement S of the form $l[\vec{e_l}] = f(r_1[\vec{e_1}],\ \ldots,\ r_k[\vec{e_k}])$ read, our ADDG construction mechanism works as follows:

(1) It extracts the iteration domain of the statement (I_S), the definition mapping (${}_SM_l^{(d)}$) and the definition domain (${}_SD_l$) of the lhs array, and the operand mapping (${}_SM_{r_i}^{(u)}$) and the operand domain (${}_SU_{r_i}$) of each rhs array $r_i,\ 1 \leq i \leq k$, and computes the dependence mapping ${}_SM_{l,r_i}$ accordingly as described above.

(2) If the elements of the array l which are used by the statements that are already read are overlapped with the elements defined by the statement S, then those elements were undefined earlier since the behaviour is in single assignment form. Therefore, we have to ensure that the elements of l defined in S are not used by the statements that are already read. For that, we find out all the read edges in G that terminates in l. We then compute the union of the ranges of the dependence mappings, r say, associated with those m read edges. If r has an overlap with ${}_SD_l$, then our construction method reports this inconsistency and stops. Otherwise, it proceeds to the next step.

(3) It next constructs the subgraph G_S of S where $G_S = \langle V_S,\ E_S\rangle$, $V_S = A_S \cup F_S$, $A_S = \{l, r_1,\ \ldots,\ r_k\}$, $F_S = \{f\}$ and $E_S = \{\langle l, f\rangle\} \cup \{\langle f, r_i\rangle,\ 1 \leq i \leq k\rangle\}$. The write edge $\langle l, f\rangle$ is associated with the statement name S and the read edge $\langle f, r_i\rangle,\ 1 \leq i \leq k$, is associated with the dependence mapping ${}_SM_{l,r_i}$. The subgraph G_S is added to G. Specifically, $V = V \cup V_S$ and $E = E \cup E_S$.

5.3 Slices

Definition 20 (Slice) *A slice is a connected subgraph of an ADDG which has an array node as its start node (having no edge incident on it), only array nodes as its terminal nodes (having no edge emanating from them), all the outgoing edges (read edges) from each of its operator nodes and exactly one outgoing edge (write edge) from each of its array nodes other than the terminal nodes.*

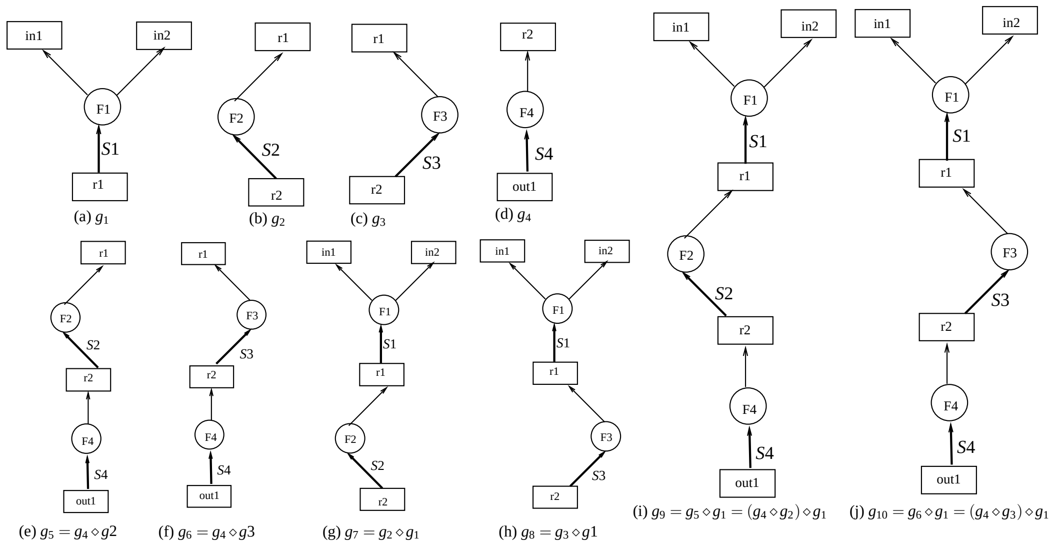

(a) g_1
(b) g_2
(c) g_3
(d) g_4
(e) $g_5 = g_4 \diamond g2$
(f) $g_6 = g_4 \diamond g3$
(g) $g_7 = g_2 \diamond g_1$
(h) $g_8 = g_3 \diamond g1$
(i) $g_9 = g_5 \diamond g_1 = (g_4 \diamond g_2) \diamond g_1$
(j) $g_{10} = g_6 \diamond g_1 = (g_4 \diamond g_3) \diamond g_1$

Figure 5.6: Slices of the ADDG in figure 5.1

```
for(i=1; i<100; i++){
   if(i < 50){
       S1: c[i] = f1(a[2i+1]);
       S2: d[i] = f2(b[i]);}
   else{
       S3: c[i] = f3(a[2i-1]);
       S4: d[i] = f4(b[i]);}}
for(i=1; i<100; i++)
       S5: e[i] = f5(c[i], d[i]);
              (a)
```

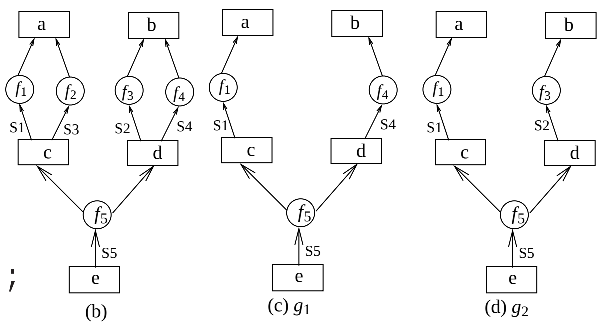


Figure 5.7: (a) A program, (b) its ADDG and (c)-(d) two slices of the ADDG

A slice represents the computation of a subset of elements of the start array node in terms of the elements of the terminal arrays. All the possible slices of the ADDG of figure 5.1, for example, are given in figure 5.6. Let g be a slice with the start array a and the terminal arrays $v_1, \ldots, v_n$, denoted as $g(a,\ \langle v_1,\ \ldots,\ v_n \rangle)$. Each of the dependence mappings ${}_gM_{a \rightsquigarrow v_i}$, $1 \leq i \leq n$, can be obtained by transitive dependence computation over a sequence of statements involved in g. The dependence mapping of the slice in figure 5.6(i), for example, can be obtained by transitive dependence over the sequence $S4, S2, S1$ of statements.

The domain of a slice is the intersection of the domains of the dependence mappings ${}_gM_{a \rightsquigarrow v_i}$, $1 \leq i \leq n$. If the intersection of these domains is empty, then it indicates that the elements of the start array of the slice are not defined in terms of the elements of the terminal arrays of that slice. Therefore, slices with empty domains can be ignored. Let us consider the program given in figure 5.7(a) and its ADDG in figure 5.7(b). Two of its possible slices are depicted in figure 5.7(c)-(d). Both the slices have start node e and two terminal nodes a and b. It may be noted that the dependence mapping ${}_{g_1}M_{e,a}$ between e and a, and the dependence mapping ${}_{g_1}M_{e,b}$ between e and b, in the slice in figure 5.7(c) are obtained from the statement sequences $(S5,S1)$ and $(S5,S4)$, respectively. Since, the iteration domain of the statements $S1$ and $S4$ are non-overlapping, the intersection of the domains of ${}_{g_1}M_{e,a}$ and ${}_{g_1}M_{e,b}$ is empty. Specifically, ${}_{g_1}M_{e,a} = \{[i] \rightarrow [2i+1] \mid 1 \leq i < 50\}$ and ${}_{g_1}M_{e,b} = \{[i] \rightarrow [i] \mid 50 \leq i < 100\}$. Therefore, this slice will be ignored. Let us now consider the slice in figure 5.7(d). For this slice, the mappings ${}_{g_2}M_{e,a}$ and ${}_{g_2}M_{e,b}$ are obtained from the statement sequences $(S5,S1)$ and $(S5,S2)$, respectively. Here, ${}_{g_2}M_{e,a} = \{[i] \rightarrow [2i+1] \mid 1 \leq i < 50\}$ and ${}_{g_2}M_{e,b} = \{[i] \rightarrow [i] \mid 0 \leq i < 50\}$. Therefore, the domain of the slice g_2 would be $0 \leq i < 50$.

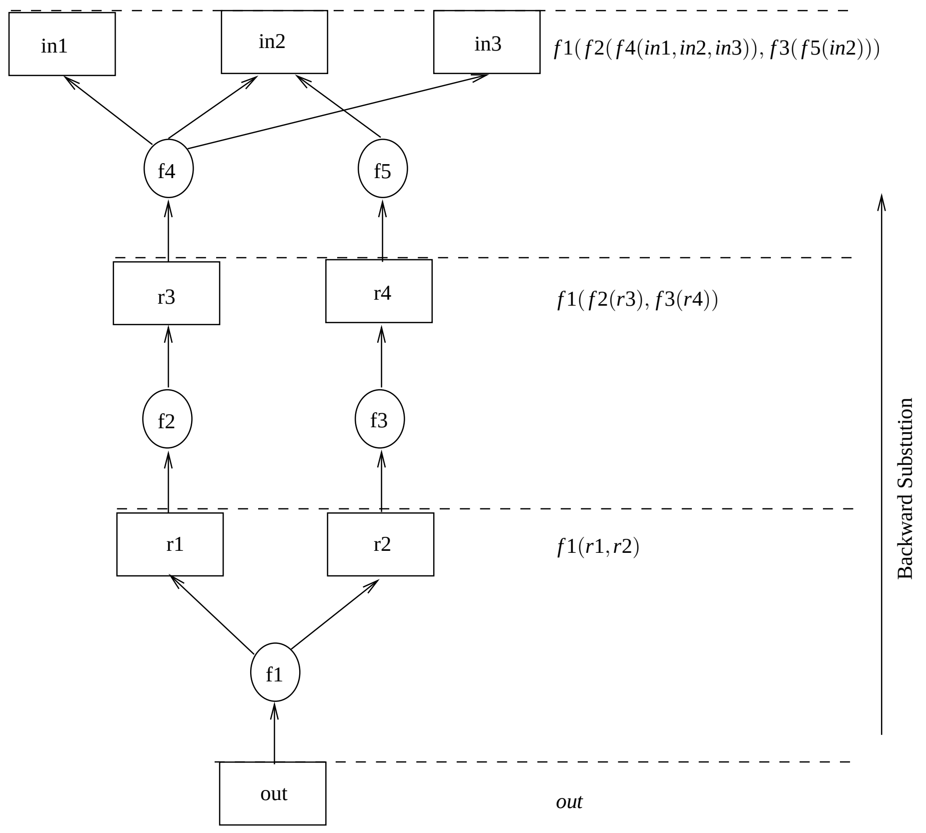


Figure 5.8: An example of computation of data transformation over a slice

The association between indices of the start array node and the terminal array nodes are captured in the dependence mappings between them in a slice. In addition, it is required to store how the start array is dependent functionally on the terminal arrays in a slice. We denote this notion as the data transformation of a slice.

Definition 21 (Data transformation of a slice g (r_g)) *It is an abstract algebraic expression e over the terminal array names of the slice such that e represents how the value of the output array of the slice depends functionally on the input arrays; the exact index expressions are abstracted out from e.*

It may be noted that that the dimensions of the arrays are left implicit and are accounted for in the dependence mapping. The data transformation r_g can be obtained by using a backward substitution method (Manna, 1974) on the slice from its output array node up to the input array nodes. The backward substitution method of finding r_g is based on symbolic simulation. The method is depicted in figure 5.8 indicating how "out" gets computed in terms of "in", "in2" and "in3".

A slice $g(a, \langle v_1,\ \ldots,\ v_n \rangle)$ is characterized by its data transformation and a list of dependence mappings between the start array a and the terminal arrays $v_1, \ldots, v_n$.

Definition 22 (Characteristic formula of a slice g (τ_g)) *The characteristic formula of a slice $g(a, \langle v_1,\ \ldots,\ v_n \rangle)$ is given as the tuple $\tau_g = \langle r_g, \langle {}_gM_{a \rightsquigarrow v_1},\ \ldots,\ {}_gM_{a \rightsquigarrow v_n} \rangle \rangle$, where ${}_gM_{a \rightsquigarrow v_i},\ 1 \le i \le n$, denotes the dependence mapping between a and v_i.*

We are interested in capturing the dependence of each output array on the input arrays of a program and its corresponding ADDG. Slices with an output array as start node and a set of input arrays as terminal nodes are meant to capture such dependencies. More specifically, we need three kinds of slices – primitive slices, component slices and IO-slices – and a composition operation of slices for this purpose. In the following, we show how to obtain the data transformation and the dependence mappings between the output array and the input arrays of a slice.

Definition 23 (Primitive slice) *Given an ADDG G of a behaviour, a primitive slice is a subgraph of G which consists of all the vertices and edges obtained from a statement of the behaviour.*

We have a primitive slice corresponding to each statement of the behaviour. For a statement S of the form $a[\ \vec{e}\ ] = f(v_1[\ \vec{e_1}\ ], \ldots, v_n[\ \vec{e_n}\ ])$ of the behaviour, the corresponding primitive slice contains the vertices a, $v_1, \ldots, v_n$, f, the write edge $\langle a, f\rangle$ and the read edges $\langle f, v_1\rangle, \ldots, \langle f, v_n\rangle$. The primitive slice also has n dependence mappings ${}_SM_{a,v_i}$, $1 \leq i \leq n$. The data transformation of a primitive slice is given by the rhs expression of the corresponding statement without the array indices; hence, for the slice corresponding to the statement S, the data transformation is $f(v_1, \ldots, v_n)$. Therefore, the characteristic formula of the primitive slice corresponding to a statement S: $a[\ \vec{e}\ ] = f(v_1[\ \vec{e_1}\ ], \ldots, v_n[\ \vec{e_n}\ ])$ is $\langle f(v_1, \ldots, v_n),\ \langle {}_SM_{a,v_1},\ \ldots,\ {}_SM_{a,v_n}\rangle\rangle$.

A slice g_1 can be composed with a slice g_2 if the start node of g_2 is a terminal node of g_1. In other words, g_1 can be composed with g_2 if the array which is defined in g_2 is one of the arrays used to define the elements of the start array of g_1. The composition of g_1 with g_2 is denoted as $g_1 \diamond g_2$. The following definition is in order.

Definition 24 (Composition of slices) *Let the characteristic formulas of two slices* $g_1(a,\ \langle b_1,\ \ldots,\ b_n\rangle)$ *and* $g_2(b_k,\ \langle c_1,\ \ldots,\ c_m\rangle)$*, for some* $k \leq n$*, be* $\tau_{g_1} = \langle f_1(b_1,\ \ldots\ , b_n),\ \langle {}_{g_1}M_{a,b_1},\ \ldots,\ {}_{g_1}M_{a,b_n}\rangle\rangle$ *and* $\tau_{g_2} = \langle f_2(c_1,\ \ldots\ , c_m),\ \langle {}_{g_2}M_{b_k,c_1},\ \ldots,\ {}_{g_2}M_{b_k,c_m}\rangle\rangle$*, respectively. Let the slice* $g(a,\ \langle b_1, \ldots, b_{k-1}, c_1,\ \ldots,\ c_m,\ b_{k+1},\ \ldots,\ b_n\rangle) = g_1 \diamond g_2$*. The characteristic formula of g is* $\tau_g = \langle f_1(b_1,\ \ldots,\ b_{k-1},\ f_2(c_1,\ \ldots\ , c_m), b_{k+1},\ \ldots,\ b_n),\ \langle {}_gM_{a,b_1},\ \ldots,\ {}_gM_{a,b_{k-1}}, {}_gM_{a,c_1},\ \ldots,\ {}_gM_{a,c_m}, {}_gM_{a,b_{k+1}},\ \ldots,\ {}_gM_{a,b_n}\rangle\rangle$*, where* ${}_gM_{a,b_i} = {}_{g_1}M_{a,b_i}$*,* $1 \leq i \leq k-1\ \wedge\ k+1 \leq i \leq n$ *and* ${}_gM_{a,c_i} = {}_{g_1}M_{a,b_k} \diamond {}_{g_2}M_{b_k,c_i}$*,* $1 \leq i \leq m$*.*

So, the data transformation of the resultant slice g is obtained by replacing the occurrence(s) of b_k in r_{g_1} by r_{g_2}. The dependence mapping between a and c_i, i.e., ${}_gM_{a,c_i}$, $1 \leq i \leq m$, is obtained from ${}_{g_1}M_{a,b_k}$ and ${}_{g_2}M_{b_k,c_i}$ using transitive dependence computation over two sequences of statements. Naturally, we ignore the resultant slice if its domain is empty.

Definition 25 (Component slice) *A slice is said to be a component slice iff (i) the slice is a primitive slice or (ii) it is obtained from composition of two component slices.*

Definition 26 (IO-slice) *A component slice is said to be an IO-slice iff its start node is an output array node and all the terminal nodes are input array nodes.*

Finally, therefore, only the IO-slices are of our interest. In general, an output array is dependent on more than one input array. Again, while a set of elements of an output array has a dependence on a set of input arrays, another set of elements of the same output array may have a different dependence on the same set of input arrays or a different set of input arrays. In general, therefore, for each output array, there may be a set of slices; (recall the restriction that a slice contains a single outgoing edge for each of its non-terminal array nodes;) each slice in the set captures a dependence of (some subset of elements of) the output array on (some subset of elements of) some input arrays.

Example 15 Let us consider the slices in figure 5.6 which are obtained from the ADDG given in figure 5.1. The ADDG is obtained from the behaviour given in example 10. The slices in figure 5.6(a)-(d) are primitive and those given in figure 5.6(i)-(j) are the two IO-slices of the ADDG. It may be noted that both the slices are from the same output node *out*1 to the same input nodes *in*1 and *in*2. Some other component slices are shown in 5.6(e)-(h). For example, the component slice in figure 5.6(e) is obtained by composing the slices in 5.6(d) and 5.6(b). □

The primitive slices are obtained from the statements of the behaviour. The component slices are obtained by composing the primitive and the component slices. The process is repeated until we obtain all the IO-slices of an ADDG; not that all the component slices need to be generated. The algorithmic form to compute all the slices of an ADDG is given as algorithm 7.

Let us now illustrate the working of the algorithm with the ADDG in figure 5.1. Different slices of this ADDG can be found in figure 5.6(a)-(j).

1. g_1, g_2, g_3, g_4 are the primitive slices. Initially, $S_G = \{g_1,\ g_2,\ g_3,\ g_4\}$.

2. In the first iteration of the while loop, the method considers g_4 which is the only slice in S_G with an output array node *out*1 as the start node. The method then composes g_4 with g_2 and g_3, respectively, and obtains the component slices g_5 and g_6, respectively. Next, it removes g_4 from S_G. After this step, g_2 and g_3 become redundant. So, they are also eliminated from S_G. After the first iteration, $S_G = \{g_1, g_5, g_6\}$.

3. In the next iteration, the method considers g_5, composes it with g_1 and obtains

Algorithm 7 Computing the IO-slices of an ADDG

```
Require: An ADDG $G$;
Ensure: $S_G$: A set of IO-slices of $G$;
1: Find all the primitive slices of $G$ and put them in $S_G$;
2: while $S_G$ contains a slice which is not an IO-slice do
3:     Take a slice $g(a,\ \langle b_1,\ \ldots,\ b_n\rangle)$ from $S_G$ which starts from an output array node $a$;
4:     Compute all the slices by composing $g$ with other slices of the form $g'(b_i,\ \langle c_1,\ldots,c_m\rangle),\ 1 \le i \le n$. If there are $l_i,\ 1 \le i \le n$, slices in $S_G$ which start from the $i^{th}$ terminal node of $g$, then we have to check $l_1 \times\ \ldots\ \times l_n$ possible resultant slices. Put the resultant slices in $S_G$. But, ignore the resultant slices with empty domains;
5:     Remove $g$ from $S_G$;
6:     Remove redundant slices from $S_G$. A slice $g'$ is redundant if its start node is not an output array node and no other slices of $S_G$ terminate at the start node of $g'$;
7: end while
```

g_9. It removes g_5 from S_G. Now, $S_G = \{g_1, g_6, g_9\}$. There is no redundant slice after this iteration.

4. In the next iteration, the method considers g_6, composes it with g_1 and obtains g_{10}. It removes g_6 from S_G. Now, g_1 is redundant. After the third step, $S_G = \{g_9,\ g_{10}\}$. Both g_9 and g_{10} are IO-slices. So, the method terminates with $S_G = \{g_9,\ g_{10}\}$. Note that the algorithm does not generate g_7 and g_8.

5.4 Equivalence of ADDGs

As discussed in the introduction, Shashidhar et al. (Shashidhar, 2008; Shashidhar et al., 2005a) have proposed an equivalence checking method for ADDG based verification of loop transformations and some data-flow transformations. In characterizing the transformation over a slice, the method proposed in Shashidhar (2008) relies on the signature of the individual paths[3] of the slice. The basic assumption in their equiv-

[3] A *path p* from an array node a_1 to an array node a_n in an ADDG is of the form $a_1 \rightarrow f_1 \xrightarrow[l_1]{} a_2 \rightarrow f_2 \xrightarrow[l_2]{} a_3 \rightarrow\ \ldots\ \xrightarrow[l_{n-2}]{} a_{n-1} \rightarrow f_{n-1} \xrightarrow[l_{n-1}]{} a_n$, where the array nodes (a_i's) and the operator nodes (f_i's) alternate, $\langle a_k, f_k\rangle,\ 1 \le k \le n-1$, are the write edges, $\langle f_k,\ a_{k+1}\rangle,\ 1 \le k \le n-1$, are the read edges in the

alence checking method is that the number of paths from the output array to the input arrays must be same in two equivalent IO-slices. Since, paths may be removed or the path signatures may be transformed significantly due to application of arithmetic transformations, computationally equivalent slices may have paths whose signatures are non-identical or have no correlation among them. Following example illustrates this fact.

```
for(k=0; k<64; k++){
        tmp1[k] = f(in3[k+1]); //S1
        tmp2[k] = in1[2k] - tmp1[k];} //S2
for(k=5; k<69; k++){
        tmp3[k] = f(in3[k-4]); //S3
        tmp4[k-5] = tmp3[k] + in2[k-3];} //S4
for(k=0; k<64; k++){
        out[k] = tmp2[k] + tmp4[k];} //S5
```

(a) Original Program

```
for(k=0; k<64; k++) {
        out[k] = in1[2k] + in2[k+2]; } //S6
```

(b) Transformed program

Figure 5.9: (a) source program; (b) transformed program

Example 16 Let us consider two program fragments given in figure 5.9. The program in figure 5.9(b) is obtained from figure 5.9(a) by loop merging and simplification of arithmetic expressions. These two programs are actually equivalent. Let us now consider their corresponding ADDGs in figure 5.10. It may be noted that both the ADDGs contain a single IO-slice. These two IO-slices have different number of paths from the output to the input arrays. Specifically, the IO-slice in figure 5.10(a) has two paths from the output array node *out* to the input array node *in*3; however, the slice in the ADDG in figure 5.10(b) has no such paths. □

ADDG and $l_i, 1, \leq i \leq n-1$, are the labels of the read edges of the path. A label l of the read edge from an operator node f to an array node a denotes that a is the l^{th} argument of the operator f.
The signature of that path is a tuple $\langle a_1, f_1, l_1, f_2, l_2, \ \ldots, \ l_{n-2}, f_{n-1}, l_{n-1}, a_n \rangle$. For example, the read edges of the ADDGs in figure 5.10 have been labelled. The signature of the path $out \rightarrow + \xrightarrow[1]{} tmp2 \rightarrow - \xrightarrow[2]{} tmp1 \rightarrow f \xrightarrow[1]{} in3$ of the ADDG given in figure 5.10(a), for example, is $\langle out, +, 1, tmp2, -, 2, tmp1, f, 1, in3 \rangle$. We have not labelled the read edges explicitly as they do not play any role in our equivalence formulation.

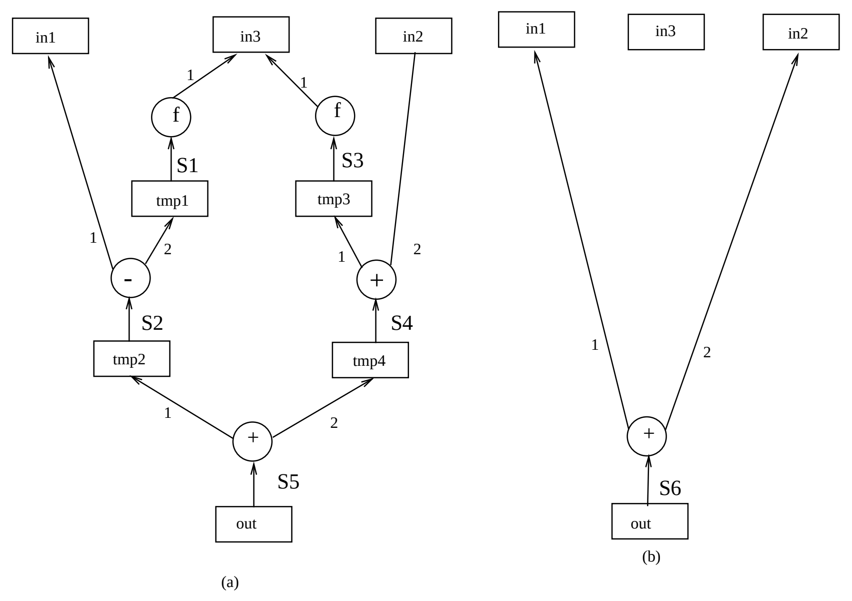


Figure 5.10: (a) ADDG of the source program; (b) ADDG of the transformed program

The above example underlines the fact that while obtaining the equivalence of a slice, *one should compare the slice as a whole rather than the individual path signatures within the slice*. In this work, we redefine the equivalence of ADDGs based on a direct slice level characterization. In addition to that, a normalization technique is incorporated in our method to represent the data transformations of the slices. Two simplification rules for normalized expression are proposed in this work. Slice level characterization and inclusion of the normalization technique and the simplification rules enable us to handle several arithmetic transformations applied along with loop transformation techniques. Let us first introduce the normalization procedure of data transformations and the simplification rules. We then formulate the equivalence problem of ADDGs.

5.4.1 Normalization of the characteristic formula of a slice

The characteristic formula of a slice consists of data transformation of the slice and a set of dependence mappings. The data transformation is an arithmetic expression over the input array names representing how the value of the output array depends functionally on the input arrays. Each dependence mapping (of the array indices) is a function from a tuple of arithmetic expressions (representing the definition domain) to another tuple of arithmetic expressions (representing the operand domain). Since a canonical form does not exist for integer arithmetic, we represent the data transformation of a slice in the same normalized form as discussed in subsection 3.2.2. For dependence mappings, we develop the following normal form.

Each of the dependence mapping can be represented as a three tuple – *index expressions of the lhs array, index expressions of the rhs array and a quantified (closed) formula defining the domain of the mapping*. The index expressions of lhs/rhs array is an ordered tuple of normalized sums where the i^{th} normalized sum represents the index expression of the i^{th} dimension of the array. The quantified formula is a conjunction of arithmetic predicates (atomic formulae) defined over the universally quantified variables. The increment/decrement operation of the universally quantified variables is captured by existentially quantified variables (as shown in definition 15). An atomic formula is of the form $SR0$, where S is a normalized sum, R is one of the relations $\{\leq,\ \geq,\ <,\ >,\ =,\ !=\}$. The dependence mappings are defined by means of productions of the following grammar.

Definition 27 *Grammar of dependence mapping:*

1. $M \rightarrow \langle L_I, R_I, D_Q \rangle$,

   / M: mapping, L_I: lhs (defined) array index expressions, R_I: rhs (used) array index expressions, D_Q: quantified formula depicting the domains of the variables in L_I and R_I */*

2. $L_I \rightarrow L_I, S \mid S$,   */* S: normalized sum */*

3. $R_I \rightarrow R_I, S \mid S$,

4. $D_Q \rightarrow \forall\exists D_Q \mid (A) \mid A$,

5. $A \rightarrow A \wedge C \mid C$,

6. $C \rightarrow SR0$,

7. $R \rightarrow\ \leq \mid \geq \mid < \mid > \mid = \mid !=$.

In addition to the above structure, any normalized formula is arranged by a lexicographic ordering of its constituent subexpressions from the bottom-most level, i.e., from the level of simple primaries. This will help us handle the algebraic transformations efficiently.

Example 17 Let us consider the dependence mapping

$M = \{[i][j][k] \rightarrow [10i+50j+k][k] \mid 0 \leq i \leq 10 \wedge \exists\alpha_i \in \mathbb{N}(i = 2\alpha_i) \wedge\ 0 \leq j \leq 50 \wedge \exists\alpha_j \in \mathbb{N}(j = 3\alpha_j) \wedge 0 \leq k \leq 20 \wedge \exists\alpha_k \in \mathbb{N}(k = 2\alpha_i)\}$.

The normalized representation of this mapping is $M = \langle D, U, Q \rangle$, where

$L_I = 1*i+0, 1*j+0, 1*k+0$,

$R_I = 10*i+50*j+1*k+0, 1*k+0$ and

$D_Q = \forall i \exists \alpha_i \forall j \exists \alpha_j \forall k \exists \alpha_k\ (1*i+0 \geq 0 \wedge 1*i+0 \leq 10 \wedge 1*i+(-2)*\alpha_i = 0 \wedge 1*j+0 \geq 0 \wedge 1*j+0 \leq 50 \wedge 1*j+(-3)*\alpha_j = 0 \wedge\ 1*k+0 \geq 0 \wedge 1*k+0 \leq 20 \wedge 1*k+(-2)*\alpha_k = 0)$. □

5.4.2 Some simplification rules for data transformations

We have proposed some simplification rules over a normalized expression. This simplification rules are used to simplify arithmetic expressions over arrays by collecting common sub-expressions. Normalization along with these simplification rules enables us handle arithmetic transformations efficiently. The simplification rules are as follows:

Rule 1: (a) For a slice g, the dependence mappings in its characteristic formula τ_g are ordered according to the occurrence of the array names in r_g.

(b) If an array name occurs more than once (as primaries) in a term of r_g, then their dependence mappings are ordered according to the lexicographic ordering of the dependence mappings.

(c) If the data transformation r_g in τ_g contains common sub-expressions with the same non-zero constant primary, then the tuple of dependence mappings corresponding to those terms are ordered according to the ordering of the corresponding dependence mappings in the tuples of the terms.

So, the rule 1 consists of three parts as shown above. An occurrence of a primary $v^{(i)}$ in an arithmetic expression signifies the i^{th} occurrence of the array variable v in it. The following example illustrates the three parts of rule 1.

Example 18 (a) Let r_g of a slice g be $1*a*b^{(1)} + 2*b^{(2)} + 1$. Let the start node (output array) of g be out. Then τ_g would be $\langle r_g, \langle\ {}_gM_{out,a},\ {}_gM_{out,b^{(1)}},\ {}_gM_{out,b^{(2)}}\ \rangle\rangle$ assuming order of the occurrences of the array names is $a \prec b^{(1)} \prec b^{(2)}$ in r_g.

(b) Let r_g of a slice g be $1*a*b^{(1)}*b^{(2)} + 2*b^{(3)} + 0$. So, the array b occurs twice in the first term of r_g. Let ${}_gM_{out,b^{(1)}} = \{[i] \rightarrow [2i] \mid 1 \le i \le N\}$ and ${}_gM_{out,b^{(2)}} = \{[i] \rightarrow [i+5] \mid 1 \le i \le 2*N\}$. Then τ_g would be $\langle r_g, \langle\ {}_gM_{out,a}, {}_gM_{out,b^{(2)}}, {}_gM_{out,b^{(1)}}, {}_gM_{out,b^{(3)}}\rangle\rangle$ since "$i+5$" $\prec$ "$2*i+0$" in the ordering of normalized sums. Instead, if ${}_gM_{out,b^{(1)}} = \{[i] \rightarrow [i+5] \mid\ 1 \le i \le N\}$ and ${}_gM_{out,b^{(2)}} = \{[i] \rightarrow [i+5] \mid 1 \le i \le 2*N\}$, then τ would be $\langle r_g,\ \langle\ {}_gM_{out,a},\ {}_gM_{out,b^{(1)}},\ {}_gM_{out,b^{(2)}},\ {}_gM_{out,b^{(3)}}\rangle\rangle$ since $1*N+0\ \prec 2*N+0$ in the ordering of normalized sums.

(c) Let r_g of a slice g be $1*a^{(1)} + 1*a^{(2)} + 0$. Let ${}_gM_{out,a^{(1)}} = \{[i] \rightarrow [i+1] \mid 1 \le i \le N\}$ and ${}_gM_{out,a^{(2)}} = \{[i] \rightarrow [2i] \mid 1 \le i \le N\}$. Then τ_g would be $\langle r_g,\ \langle\ {}_gM_{out,a^{(1)}}, {}_gM_{out,a^{(2)}}\rangle\rangle$. □

Rule 2: In the data transformation of a slice, the occurrences of a common sub-expression are collected together if the dependence mappings from the output array to each of the (input) arrays involved in the occurrences of the sub-expression are equal. Let an expression e consist of p occurrences of the sub-expression e_s. Let the sub-expression involve k input arrays, $in_1,\ \ldots,\ in_k$. Formally, let this be denoted as $e(e_s^{(1)}(in_1,\ \ldots,\ in_k),\ \ldots,\ e_s^{(p)}(in_1,\ \ldots,\ in_k))$. These p sub-expressions can be collected iff $\forall j \forall l \forall m,\ 1 \le j \le k,\ 1 \le l,\ m \le p\ M^{(l)}_{out,\ in_j} = M^{(m)}_{out,\ in_j}$, where $M^{(l)}_{out,\ in_j}$ ($M^{(m)}_{out,\ in_j}$) denotes the dependence mapping from the index space of the array out (start node of the slice) to that of the input array in the l^{th} (m^{th}) sub-expression. If the collection of the sub-expressions cancels out through symbolic computation, then remove all the dependence mappings $M^{(l)}_{out,\ in_j}$, $1 \le l \le p$, $1 \le j \le k$, from the output array to each of the (input) arrays from the characteristic formula of the slice.

The following example illustrates the rule 2.

Example 19 Let us consider, for example, that the data transformation of a slice

$g(a,\ \langle x,z\rangle)$ is $3x\ +\ 5z\ -\ 3x$, where x and z be two input arrays and a be the output array. Let the dependence mapping from the output array a of the slice to x corresponding to the first sub-expression $3x$ be ${}_gM^{(1)}_{a\leadsto x}$ and the same for x corresponding to the second sub-expression $3x$ be ${}_gM^{(2)}_{a\leadsto x}$. This formula is reduced to $5z$ if the dependence mapping ${}_gM^{(1)}_{a\leadsto x}\ =\ {}_gM^{(2)}_{a\leadsto x}$. Similarly, the formula $3xy\ +\ 4z\ +\ 7xy$ is reduced to $10xy+4z$ if the dependence mappings from output array to x are the same for both occurrences of x (in $3xy$ and $7xy$) and the dependence mappings from output array to y are the same for both occurrences of y (in $3xy$ and $7xy$). $\square$

These simplification rules enable us to represent two arithmetically transformed expressions uniformly. Let us consider the behaviours in figure in 5.9 and their corresponding ADDGs in figure 5.10. Both the ADDGs contain a single slice. The data transformation of the slice, g_1 say, in the ADDG in figure 5.10(a) is $r_{g_1} = in1 + f(in3^{(1)}) - f(in3^{(2)}) + in2$. The two dependence mappings from out to $in3$ are ${}_{g_1}M_{out\leadsto in3^{(1)}} = \{[k] \rightarrow [k+1] \mid 0 \leq k \leq 64\}$ and ${}_{g_1}M_{out\leadsto in3^{(2)}} = \{[k] \rightarrow [k+1] \mid 0 \leq k \leq 64\}$ in figure 5.10(a). Here, the dependence mappings for both the occurrences of $in3$ in the data transformation are the same. Hence, the data transformation r_{g_1} is reduced to $in1 + in2$. On the other hand, the data transformation of the slice, g_2 say, in the ADDG in figure 5.10(b) is $in1 + in2$. It can be shown that the dependence mapping from out to $in1$ and dependence mapping from out to $in2$ are the same in both the slices. Hence, the slice g_1 of the ADDG in figure 5.10(a) and the slice g_2 of the ADDG in figure 5.10(b) are equivalent. Also, the dependence mappings ${}_gM^{(1)}_{out\leadsto in3}$ and ${}_gM^{(2)}_{out\leadsto in3}$ are removed from the characteristic formula of this slice since they now become redundant.

Let us again consider the source behaviour in figure 5.9(a). Let the statement $S3$ be modified as $tmp3[k] = f(in3[k])$ in that behaviour. Then, the dependence mapping would be ${}_{g_1}M_{out\leadsto in3^{(1)}} = \{[k] \rightarrow [k+5] \mid 0 \leq k \leq 64\}$ and ${}_{g_1}M_{out\leadsto in3^{(2)}} = \{[k] \rightarrow [k+1] \mid 0 \leq k \leq 64\}$. Clearly, they are not the same. Therefore, the data transformation $r_{g_1} = in1 + f(in3^{(1)}) - f(in3^{(2)}) + in2$ for this modified behaviour cannot be simplified to $in1 + in2$. It may be noted that the behaviour in figure 5.9(a) with the modification stated above is no more equivalent with the behaviour in figure 5.9(b) since the data transformation of g_1 and g_2 are not the same now.

5.4.3 Equivalence problem formulation

Let G_S be the ADDG corresponding to an input behaviour and G_T be the ADDG corresponding to the transformed behaviour obtained from the input behaviour through loop and arithmetic transformations. In the following the definition of equivalence between the two ADDGs is evolved.

Definition 28 (Matching IO-slices) *Two IO-slices g_1 and g_2 of an ADDG G are said to be matching, denoted by $g_1 \approx g_2$, if the data transformations of both the slices are equivalent.*

Let the characteristic formula of $g_i(a, \langle v_1, \ldots, v_l \rangle)$, $i = 1, 2$, be $\langle r_{g_i}, \langle {}_{g_i}M_{a \rightsquigarrow v_1}, \ldots, {}_{g_i}M_{a \rightsquigarrow v_l} \rangle \rangle$, where a is an output array and $v_1, \ldots, v_l$ are the input arrays. These two slices are matching slices if r_{g_1} and r_{g_2} are equivalent. Due to single assignment form of the behaviour, the domain of the dependence mapping between the output array a and the input array v_j in the slices g_1 and g_2 of the ADDG G, however, are non-overlapping.

Definition 29 (IO-slice class) *An IO-slice class is a maximum set of matching IO-slices.*

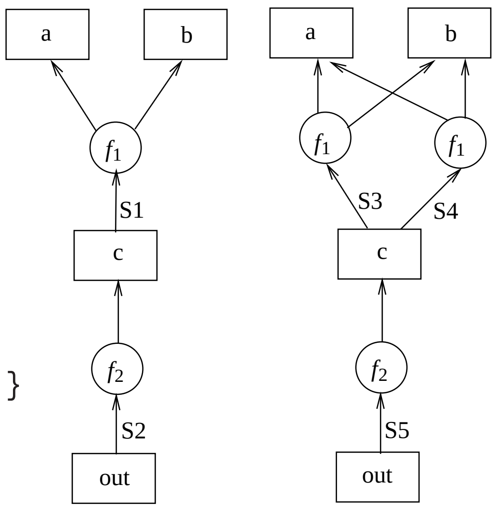
Let a slice class be $C_g(a, \langle v_1, \ldots, v_l \rangle) = \{g_1, \ldots, g_k\}$ where each slice involves l input arrays $v_1, \ldots, v_l$ and the output array a. Let the characteristic formula of the member slice g_i, $1 \leq i \leq k$, be $\langle r_{g_i}, \langle {}_{g_i}M_{a \rightsquigarrow v_1}, \ldots, {}_{g_i}M_{a \rightsquigarrow v_l} \rangle \rangle$. Due to single assignment form of the behaviour, the domain of the dependence mappings between the output array a and the input array v_m, $1 \leq m \leq l$, in the slices of C_g must be non-overlapping, that is ${}_{g_i}M_{a \rightsquigarrow v_m}$ and ${}_{g_j}M_{a \rightsquigarrow v_m}$, $1 \leq i, j \leq k$, are non-overlapping. The domain of the dependence mapping ${}_{C_g}M_{a \rightsquigarrow v_m}$ from a to v_m over the entire class C_g is the union of the domains of ${}_{g_i}M_{a \rightsquigarrow v_m}$, $1 \leq i \leq k$. So, the characteristic formula of the slice class C_g is $\langle r_{C_g}, \langle {}_{C_g}M_{a \rightsquigarrow v_1}, \ldots, {}_{C_g}M_{a \rightsquigarrow v_l} \rangle \rangle$, where r_{C_g} is the data transformation of any of the slices in C_g and ${}_{C_g}M_{a \rightsquigarrow v_m}$, $1 \leq m \leq l$, is

$${}_{C_g}M_{a \rightsquigarrow v_m} = \bigcup_{1 \leq i \leq k} {}_{g_i}M_{a \rightsquigarrow v_m}$$

Therefore, a slice class can be visualized as a single slice. The characteristic formula of a slice class C_g is denoted as τ_{C_g}.

```
for(k = 0; k <= 100; k++)
    S1: c[k] = f1(a[2k], b[k+1]);
for(i=0; i<=50; i++)
   for(j=0; j<=50; j++)
       S2: out[i][j] = f2(c[i+j]);
     (a) original program

for(k = 0; k <= 100; k +=2){
    S3: c[k] = f1(a[2k], b[k+1]);
    S4: c[k+1] = f1(a[2k+2], b[k+2]);}
for(i=0; i<=50; i++)
   for(j=0; j<=50; j++)
       S5: out[i][j] = f2(c[i+j]);
     (b) transformed program
```



(c) ADDG of (a) (d) ADDG of (b)

Figure 5.11: An example for matching slices and slice class

Definition 30 (IO-slice class equivalence) *A slice class C_1 of an ADDG G_S is said to be equivalent to a slice class C_2 of G_T, denoted as $C_1 \simeq C_2$, iff $\tau_{C_1} = \tau_{C_2}$.*

Let us also denote the slice class C_2 as the corresponding slice class of C_1 when $C_1 \simeq C_2$.

Definition 31 (Equivalence of ADDGs:) *An ADDG G_S is said to be equivalent to an ADDG G_T iff for each IO-slice class C_S in G_S, there exists an IO-slice class C_T in G_T such that $C_S \simeq C_T$, and vice-versa.*

Example 20 Let us consider the behaviours in figure 5.11. The transformed behaviour in figure 5.11(b) is obtained by loop unrolling of the first loop body of the source program in figure 5.11(a). The corresponding ADDGs are depicted in figure 5.11(c) and figure 5.11(d). There is only one IO-slice (i.e., the ADDG itself), g_1 say, in the ADDG in figure 5.11(c). For the slice g_1,

$r_{g_2} = f_2(f_1(a,b))$,

${}_{g_1}M_{out,a} = \{[i,j] \rightarrow [2i+2j] \mid 0 \leq i \leq 50 \wedge 0 \leq j \leq 50\}$ and

$_{g_1}M_{out,b} = \{[i,j] \rightarrow [i+j+1] \mid 0 \leq i \leq 50 \wedge 0 \leq j \leq 50\}$.

The ADDG in figure 5.11(d) consists of two IO-slices, g_2 and g_3, say, where the IO-slice g_2 consists of the statements $S5$ and $S3$ and the IO-slice g_3 consists the statements $S5$ and $S4$. For the slice g_2,

$r_{g_2} = f_2(f_1(a,b))$,

$_{g_2}M_{out,a} = \{[i,j] \rightarrow [2i+2j] \mid 0 \leq i \leq 50 \wedge 0 \leq j \leq 50 \wedge \exists \alpha \in \mathbb{Z}(2\alpha = i+j)\}$ and

$_{g_2}M_{out,b} = \{[i,j] \rightarrow [i+j+1] \mid 0 \leq i \leq 50 \wedge 0 \leq j \leq 50 \wedge \exists \alpha \in \mathbb{Z}(2\alpha = i+j+2)\}$.

For the slice g_3,

$r_{g_3} = f_2(f_1(a,b))$,

$_{g_3}M_{out,a} = \{[i,j] \rightarrow [2i+2j] \mid 0 \leq i \leq 50 \wedge 0 \leq j \leq 50 \wedge \exists \alpha \in \mathbb{Z}(2\alpha = i+j-1)\}$ and

$_{g_3}M_{out,b} = \{[i,j] \rightarrow [i+j+1] \mid 0 \leq i \leq 50 \wedge 0 \leq j \leq 50 \wedge \exists \alpha \in \mathbb{Z}(2\alpha = i+j+1)\}$.

It may be noted that the data transformations of both g_2 and g_3 are same. So, the IO-slices g_2 and g_3 are matching IO-slices and they form a slice class C_g in the ADDG in figure 5.11(d). For the slice class C_g,
$r_{C_g} = f_2(f_1(a,b))$,
$_{C_g}M_{out,a} = {}_{g_2}M_{out,a} \cup {}_{g_3}M_{out,a} = \{[i,j] \rightarrow [2i+2j] \mid 0 \leq i \leq 50 \wedge 0 \leq j \leq 50\}$ and
$_{C_g}M_{out,b} = {}_{g_2}M_{out,b} \cup {}_{g_3}M_{out,b} = \{[i,j] \rightarrow [i+j+1] \mid 0 \leq i \leq 50 \wedge 0 \leq j \leq 50\}$.
Therefore, g_1 is equivalent to C_g. Hence, the ADDGs in figure 5.11(c) and in figure 5.11(d) are equivalent. $\square$

The equivalence checking method is given as algorithm 8. The basic steps of the algorithm are as follows:

1. We first obtain the possible IO-slices of an ADDG with their characteristic formulae comprising their dependence mappings and the data transformations. The data transformation of a slice will be represented as a normalized expression. The algebraic transformations based on associativity and commutativity and distributivity will be taken care of by the normalization process itself.

Algorithm 8 Equivalence Checking Between two ADDGs

```
1: /* Input: Two ADDGs $G_S$ and $G_T$;
   Output: Whether $G_S$ and $G_T$ are equivalent or not; */
2: Find the set of IO-slices in each ADDG. Find the characteristic formulae of the slices;
3: Use arithmetic simplification rule to the data transformation of the slices of $G_S$ and $G_T$;
4: Obtain the slice classes ensuring non-intersection of the dependence mapping domains of the constituent slices and their characteristic formula in each ADDG; Let $\mathcal{C}_{G_S}$ and $\mathcal{C}_{G_T}$ be the respective sets of slice classes in both the ADDGs;
5: for each slice class $g_1$ in $\mathcal{C}_{G_S}$ do
6:     $g_k = findEquivalentSliceClass(g_1,\ \mathcal{C}_{G_T})$;
7:     /* this function returns the equivalent slice of $g_1$ in $\mathcal{C}_{G_T}$ if found; otherwise returns NULL; */
8:     if $g_k = NULL$ then
9:         Report "ADDGs may not be equivalent;" exit (failure);
10:    end if
11: end for
12: Repeat the above loop by interchanging $G_S$ and $G_T$;
13: Report "ADDGs are equivalent;" exit (success);
```

2. We then apply the simplification rule on the slice data transformations. The effect of other arithmetic transformations like, common sub-expression elimination, constant folding, arithmetic expression simplification, etc. will be handled by the simplification rule. After simplification of the data transformations, two equivalent slices have the same characteristic formula.

3. We then form slice classes by collecting the matching slices in an ADDG. We now compute the characteristic formula of each slice class from its member slices.

4. We now establish equivalence between slice classes of the ADDGs. The function $findEquivalentSliceClass$ in algorithm 8 is used for this purpose. For each slice class, this function tries to find its equivalent slice class in the other ADDG. Corresponding to each slice class, the function uses the data transformation to find the matching slice class in the other ADDG first and then compare the re-

spective dependence mappings.

```
for(k=0; k<64; k++){
        tmp1[k] = b[2k+1] + c[2k];
        tmp2[k] = a[k] × tmp1[k]; }
for(k=5; k<69; k++){
        tmp3[k] = a[k-5] - c[2k-10];
        tmp4[k] = tmp3[k] × b[2k-9]; }
for(k=0; k<64; k++){
        tmp5[k] = a[k] × c[2k];
        out[k] = tmp2[k] - tmp4[k] + tmp5[k]; }
```

(a) original behaviour

```
for(k=0; k<64; k++){
        tmp[k] = 2a[k] + b[2k+1];
        out[k] = c[2k] × tmp[k];}
```

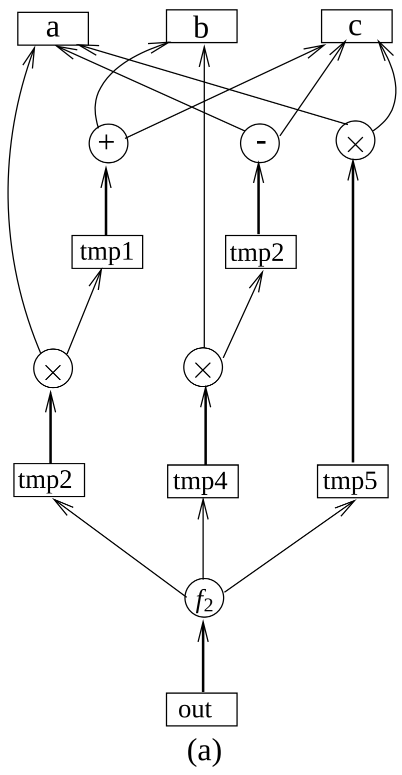
(b) transformed behaviour

Figure 5.12: (a) original behaviour; (b) transformed behaviour

5.5 A case study

Let us consider the source behaviour of figure 5.12(a) and its corresponding transformed behaviour of figure 5.12(b). In this example, a, b and c are three input arrays and *out* is the output array. The transformed behaviour is obtained from the source behaviour by application of loop fusion (Bacon et al., 1994) along with distributive transformations and arithmetic transformation. The ADDG of the original behaviour is depicted in figure 5.13(a) and the same for the transformed behaviour is depicted in figure 5.13(b).

It may be noted that each of the ADDGs has only one slice. Let the slices of the source behaviour and the transformed behaviour be denoted as s and t, respectively. Our method first extracts the data transformation of the slice and the dependence mapping of each path of the slice s and t. The data transformation of the slice s, i.e., r_s is $a(b+c)-b(a-c)+ac$ and that of the slice t, i.e., r_t is $c(2a+b)$; The normalized representation of r_s is $1*a*b\ +\ (-1)*a*b\ +\ 1*a*c\ +\ 1*a*c\ +\ 1*b*c\ +\ 0$. In this normalized expression, the first and the second terms can be eliminated as the dependence mappings from the output array *out* to the arrays a and b are the same in them. In



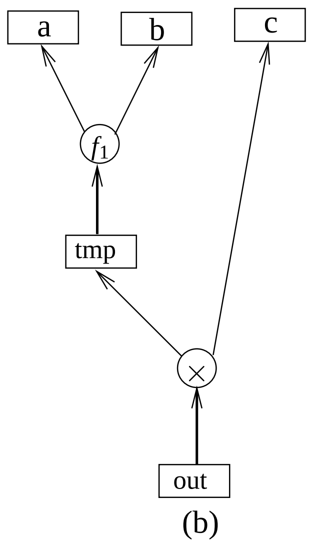


Figure 5.13: (a) ADDG of the original program; (b) ADDG of the transformed program

particular, the dependence mapping from *out* to *a* is $M_{out,a} = \{[k] \rightarrow [k] \mid 0 \leq k < 64\}$ and the dependence mapping from *out* to *b* is $M_{out,b} = \{[k] \rightarrow [2k+1] \mid 0 \leq k < 64\}$ in both the terms. Similarly, the third and the fourth terms of the normalized expression have the same dependence mappings from *out* to *a* and the dependence mappings from *out* to *c*. In particular, the dependence mappings from *out* to *a* and from *out* to *c* are $M_{out,a} = \{[k] \rightarrow [k] \mid 0 \leq k < 64\}$ and $M_{out,c} = \{[k] \rightarrow [2k] \mid 0 \leq k < 64\}$, respectively. So, these two terms can be collected. Therefore, after application of our simplification rules, r_s becomes $2*a*c\ +\ 1*b*c\ +\ 0$. The normalized representation of r_t is $2*a*c\ +\ 1*b*c\ +\ 0$. Therefore, $r_s \simeq r_t$. After simplification, the data transformations of the slices consist of three input arrays including two occurrences of c. So, we need to check four dependence mappings, each one from *out* to the input arrays a, b, $c^{(1)}$ and $c^{(2)}$. The respective dependence mappings are $M_{out,a} = \{[k] \rightarrow [k] \mid 0 \leq k < 64\}$, $M_{out,b} = \{[k] \rightarrow [2k+1] \mid 0 \leq k < 64\}$, $M_{out,c^{(1)}} = \{[k] \rightarrow [2k] \mid 0 \leq k < 64\}$ and $M_{out,c^{(2)}} = \{[k] \rightarrow [2k] \mid 0 \leq k < 64\}$, respectively in the slice s. It can be shown that $M_{out,a}$, $M_{out,b}$, $M_{out,c^{(1)}}$ and $M_{out,c^{(2)}}$ in slice t are the same as those in the slice s. So, the slices s and t are equivalent. Hence, the

ADDGs are equivalent.

5.6 Correctness and complexity

Two behaviours are said to be equivalent iff for every input, both the behaviours produce the same set of outputs. The method presented here examines the equivalence of the ADDGs G_S and G_T obtained from the input (source) behaviour and the transformed behaviour. For soundness of our method, we need to show that G_S and G_T are indeed equivalent when the algorithm ascertains them to be so. Following theorem captures the fact.

Theorem 19 (Soundness) *If the algorithm 8 terminates successfully in step 13, then two ADDGs G_S and G_T are equivalent.*

Proof: Let there be l IO-slice classes $g_{s_1}, \ldots, g_{s_l}$ in the set $\mathcal{C}_{S_g}$ of slice classes of G_S which start from the output array o. Our method finds that the set $\mathcal{C}_{T_g}$ of IO-slice classes in ADDG G_T comprises exactly l slice classes $g_{t_1}, \ldots, g_{t_l}$, say which also start from array o such that $g_{s_i} \simeq q_{t_i}, 1 \leq i \leq l$. Each slice class g_{s_i} defines a part of the array o. Since the behaviours are considered to be in single assignment form, the parts of o defined by $g_{s_i}, 1 \leq i \leq l$, are non overlapping. Also, there is no other slice class that starts from o in $\mathcal{C}_{S_g}$. Therefore $g_{s_i}, 1 \leq i \leq l$, together define the whole array. Moreover, there cannot be any more slice class other than $g_{t_1}, \ldots, g_{t_l}$ in $\mathcal{C}_{T_g}$ starting from o; otherwise, our method will find a possible non-equivalence of that slice class in step 12 of algorithm 8. In the same way, we can show that for all other output arrays, there are equal number of slice classes in $\mathcal{C}_{T_g}$ and in $\mathcal{C}_{T_g}$ which start from that array. Therefore, if the algorithm 8 proceeds up to step 12, then both the ADDGs have equal number of slice classes and there is a one-to-one correspondence between the slice classes of the ADDGs. Let the ADDG G_S have slice classes $g_{s_1}, \ldots, g_{s_n}$ and the ADDG G_T has slice classes $g_{t_1}, \ldots, g_{t_n}$ and $g_{s_i} \simeq q_{t_i}, 1 \leq i \leq n$. Let us assume that the ADDG G_S is not equivalent to the ADDG G_T even if there is a one-to-one correspondence between the slices of $\mathcal{C}_{T_g}$ and $\mathcal{C}_{T_g}$. In the following, the assumption is proved to be wrong by contradiction.

The two ADDGs G_S and G_T are not equivalent in the following cases.

(i) The final values of some elements of one output array are different in two ADDGs: As discussed above, each element of any output array is defined through a slice. Since we are considering the same set of elements of an output array, they must be associated with some corresponding slice class of G_S and G_T. Let us assume that the slice classes g_{s_i} and g_{t_i} be associated with those elements of the output array in G_S and G_T, respectively. Since, transformations do not match for these elements, the data transformations of g_{s_i} and g_{t_i} should be different. However, step 6 of algorithm 8 already ensures that $r(g_{s_i}) \simeq r(g_{t_i})$ (contradiction).

(ii) The association between the index of an output array, o say, and an input array, *in* say, is not the same for some elements of an output array: As reasoned above in case (i), those elements of o must be associated with the corresponding slice class of G_S and G_T. Let us assume that the slice classes g_{s_i} and g_{t_i} be associated with those elements of o in G_S and G_T, respectively. Since, the association between the index of o and that of the array *in* is not the same for those elements of o, the mappings ${}_{g_{s_i}}M_{o,in}$ and ${}_{g_{t_i}}M_{o,in}$ are not the same. However, step 6 of algorithm 8 already ensures that ${}_{g_{s_i}}M_{o,in} \simeq {}_{g_{t_i}}M_{o,in}$ (contradiction).

Hence, the ADDGs G_S and G_T are indeed equivalent. □

Given the undecidability results of the equivalence problem of flow chart schemas, completeness, however, is unattainable (Howden, 1987; Manna, 1974).

5.6.1 Complexity

Complexity of algorithm 7 (finding slices in an ADDG)

In algorithm 7, we first find all the primitive slices of the behaviour. Let the number of statements be p in the bahaviour. So, the complexity of this step is $O(p)$. A primitive slice is not composed with a component slice in each iteration of a while loop. This loop terminates when all the IO-slices of the ADDG are obtained. We need to find the complexity of this loop and also the number of IO-slices in an ADDG. Let us consider the control flow graph (CFG) of the behaviour for this purpose. Our objectve is to find the correlation between the number of IO-slices in the ADDG with the number of paths in the CFG. Let us denote a specie of code of the form $\texttt{if(c)\{//if-body\}else\{//else-body\}}$ as a branching block with two branches. A sequence of statements (without any conditional statement) is considered as a branching block with one branch. Let us ignore the back edges (corresponding to the loops) in the CFG. In such a case, the CFG is nothing but a sequence of branching blocks. Let the number of branching blocks in the CFG be n and the maximum branches in a branching block be k. Therefore, the maximum number of possible execution paths from the start node to the terminal nodes of the CFG be k^n. Let us now relate the CFG (without back edges) with the ADDG. If the array A is being defined in one branch of a branching block of CFG, then that branch creates an outward (write edge) from the array node A in the ADDG. In general, more than one array may be defined in one branch of a branching block. Let the maximum number of arrays defined in a branch be x. Therefore, each branch of a branching block of the CFG creates at most x write edges from the corresponding array nodes in the ADDG. In the worst case, the same x arrays are defined in each of the branch of a branching block and all the x arrays defined within a branch form a data dependent chain. Also, the arrays defined in one branching block may depend on the arrays defined in the preceding blocks in the worst case. So, the maximum height of the ADDG in the worst case is $n \times x$. Also, there may be maximum k number of write edges from an array node in the ADDG. Therefore, in the worst case, the number of slices in an ADDG is $O(k^{n \times x})$. In the best case, all the branching blocks are not data dependent and one array is defined only in one branch of the branching blocks. So, in the best case, the height of the ADDG is one and the maximum possible slices in an ADDG be $n \times k \times x$. To compute a slice in an ADDG with height h, we need $O(h)$ iterations of the algorithm 7. Therefore, the complexity of the algorithm 7 for finding the slices in an ADDG is $O(n \times x \times k^{n \times x})$ in the worst case and is $O(n \times k \times x)$ in the best case.

Complexity of the normalization procedure

Let us now identify the complexity of normalizing a formula F. If $\|F\|$ be the length of the formula (i.e., the number of variable occurrences plus that of the operations in F), then the complexity of normalization of F is $O(\|F\|^2)$ due to multiplication of normalized sums. (For all other operations, it is linear in $\|F\|$). The complexity in comparing two normalized formulas is $O(\|F\|)$, where $\|F\|$ is the length of maximum of two formulas. Let the number of arrays in the behaviour be a.

Let us now obtain the complexity of the simplification rules (of subsection 5.4.2).

1.(a) The number of arrays in a formula be $O(a)$. So, the complexity of ordering the dependence mapping of a slice is $O(a \times \|F\|)$.

1.(b) Each array may occur multiple times in a term of a normalized sum. Ordering multiple occurrences of an array in a term requires $O(\|F\| \times log(\|F\|))$ time. So, ordering multiple occurrences of the arrays in a term requires $)(a \times \|F\| \times log(\|F\|))$.

1.(c) The number of arrays in a term is $O(a)$. The complexity of comparing two common sub-expressions (terms) is $O(a^2\|F\|)$.

2. In this case, we have to compare two common sub-expressions. So, the complexity of this step also $O(a^2\|F\|)$.

Complexity of algorithm 8 (equivalence checking between two ADDGs)

Let the number of slices in the two ADDGs G_S and G_T be s_1 and s_2, respectively. So, normalization of the data transformations and the index expressions for the slice $s_1(s_2)$ requires $O(s_1 \times a^2\|F\|)$ ($O(s_2 \times a^2\|F\|)$) time. Each slice needs to compare with every other slices of G_S (G_T) to find the slice classes in G_S (G_T). So, the complexity of finding the slice classes in G_S (G_T) is $O(s_1^2 \times \|F\|)$ ($O(s_2^2 \times \|F\|)$). To find the equivalent of a slice class of G_S in G_T, we have to compare the data transformations and the dependence mappings of each slice of G_S with every slice class of G_T. Therefore, the complexity of equivalence checking between the slice classes of G_S and G_T is $O(s_1 \times s_2 \times \|F\|)$. So, the overall complexity of the algorithm 8, is $O(s \times a^2\|F\| + s^2 \times \|F\|)$, where s is maximum of s_1 and s_2. Considering the best case and worst case values of s, the complexity of equivalence checking is $O(k^{n \times x} \times a^2 \times \|F\| + k^{2 \times n \times x} \times \|F\|)$ in the worst case and $O(n \times k \times x \times a^2 \times \|F\| + (n \times k \times x)^2 \times \|F\|)$ in the best case.

For computing the transitive dependence and comparing the two dependence mappings, we have used the Omega calculator (Kelly et al., 2008). It has one component called Omega test which is a system for manipulating sets of affine constraints over integer variables. The Omega test framework is an integer programming solver for the Presburger arithmetic based on Fourier-Motzkin variable elimination. The index expressions in the dependence mapping are affine over integer variables. Therefore, Omega test can be used for our purpose. The deterministic upper bound on the time required to verify Presburger formulas is $2^{2^{2^n}}$, where n is the length of the formula. However, in practice, the Omega test is reasonably efficient for simplifying and verifying Presburger formulas.

5.7 Error diagnosis

In case of non-equivalence of two behaviours, our method can provide some additional information to the users to help them localize the possible causes of non-equivalence. In the following, we discuss some such situations.

During computing IO-slices:

- For an output array a, if the union of domains of all the IO-slices starting from a is not equal to the union of the definition domains of a, it means some of the elements of a are not actually defined in terms of the input arrays in the behaviour. In this case, even if the transformed behaviour is found to be equivalent to the original behaviour, the latter is itself inconsistent. After computing all the IO-slices of an ADDG, we can do this additional consistency checking in algorithm 7. In such cases of inconsistency, we can report the output array name, the set of statements involved in those IO-slices and the domain of the undefined elements of the output array.

During Equivalence Checking:

- The data transformation of one slice class of one ADDG does not match with any of the slice classes of the other ADDG. In this case, our method can report the slice, data transformation of it and the set of statements it involves. In addition to that, we can add a heuristic to find slice(s) with nearest data transformation in the other ADDG. The heuristic may be based on the finding longest common subsequence (LCS) (Cormen et al., 2001) of the data transformations of the two slices.

- Corresponding to a slice class *s* of an ADDG, our method finds a slice class *t* in the other ADDG with data transformation same as that of *s* but the index expressions for some input arrays do not match in the corresponding dependence mappings. In this case, our method can report the slices, the set of statements they involve, precise array names whose indices mismatch and their respective index expressions.

- Corresponding to a slice class *s* of an ADDG, our method finds a slice class *t* in the other ADDG with data transformation same as that of *s*. Also, the association rules in the corresponding dependence mappings are same. However, the domains of some of the corresponding dependence mappings of *s* and *t* are not the same. In this case also, our method can report the slices, the set of statements they involve and the array names it involved.

5.8 Experimental results

	nesting	*loops*		*arrays*		*slices*		*Exec time (sec)*		
Cases	*depth*	*src*	*trans*	*src*	*trans*	*src*	*trans*	*equiv*	*not-equiv1*	*not-equiv2*
(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)	(11)
SOB1	2	3	1	4	4	1	1	01.53	0.62	0.75
SOB2	2	3	3	4	4	1	1	11.21	0.72	0.46
WAVE	1	1	2	2	2	4	4	07.05	0.73	0.59
LAP1	2	1	3	2	4	1	1	02.31	0.43	0.32
LAP2	2	1	1	2	2	1	2	07.58	0.26	0.24
LAP3	2	1	4	2	4	1	2	02.12	1.14	1.13

Table 5.1: Results for several benchmarks

The method has been implemented in C language and run on a 2.0 GHz Intel® Core™2 Duo machine. Our tool first extracts the ADDGs from the source and the transformed behaviour written in C and applies the method to establish the equivalence between them. For the dependence mappings of the slices, our method relies on the Omega calculator (Kelly et al., 2008). The method has been tested on several instances of equivalence checking problems obtained manually from the sobel edge detection (SOB), debaucles 4-coefficient wavelet filter (WAVE) and Laplace algorithm to edge enhancement of northerly directional edges (LAP). To create the test cases, we have considered variety of loop transformations and arithmetic transformations. Specifically, test cases have been obtained by applications of following loop and arithmetic transformations: (i) SOB1: loop fusion, commutative and distributive, (ii) SOB2: loop reorder, commutative and distributive, (iii) WAVE: loop un-switching and commutative, (iv) LAP1: expression splitting and loop fission, (v) LAP2: loop unrolling, commutative and distributive, and (vi) LAP3: loop spreading, commutative and remaining. The maximum nesting depth of the loops, numbers of loop bodies, arrays and slices in the source and the transformed behaviours are shown in columns two to eight in table 5.1. The execution time of our tool for these test cases are tabulated in column nine of the table. In the last two cases, the number of slices differs from the source ADDG to the transformed ADDG, the method successfully forms the slice classes and establishes their equivalence. In all of the cases, our method was able to establish the equivalence in less than twelve seconds. It may be noted that the method reported in (Shashidhar et al., 2005a) fails in the cases of SOB1, SOB2, LAP2 because of application of distributive transformations.

In our second experiment, we take the source and the transformed behaviours of the previous experiment. We, however, intentionally change the index expressions of some of the arrays or limits of the loops in the transformed behaviours. As a result, some dependence mappings (involving those arrays) do not match with the original behaviour. Similarly, we change the rhs expressions of some statements of the transformed behaviours to create another set of erroneous test cases. As a result, the data transformations of some of the slices do not match with the corresponding slices of the original behaviour. The execution time of our tool for these two cases are tabulated in columns 10 and 11, respectively. Our tool is able to find the non-equivalence of in all the cases in less than two seconds as shown in table 5.1. One important aspect of our implementation is that it provides some additional information to the users to help them localize the possible causes of non-equivalence in the case of non-equivalence of ADDGs as discussed in subsection 5.7.

5.9 Conclusion

This chapter presents a verification method of loop transformations and arithmetic transformation techniques applied on loop and array intensive applications in the multimedia and signal processing domain. An ADDG based equivalence checking method is proposed for this purpose. The method relies on normalization of arithmetic expressions and simplification rules to handle arithmetic transformations applied along with loop transformations. Unlike many other reported techniques, our method is strong enough to handle arithmetic transformations like associative, commutative, distributive, arithmetic expressions simplifications, common sub-expressions elimination, constant folding, copy propagation, etc. Correctness and complexity of the method are also discussed. Experimental results have shown the efficiency of the method. The future scope of work includes identification of the simplification rules to handle sophisticated arithmetic transformations like operator strength reduction. To find the scope of application of our normalization techniques to widening based approach (Verdoolaege et al., 2009) which can handle non-uniform recurrence can also be explored in future.

Chapter 6

Verification of Parallelizing Transformations

6.1 Introduction

Modern day embedded system designs consist of multiprocessor systems to meet the contrasting needs of high computational performance with low power consumption. To deploy suitably in a multiprocessor system, the initial sequential behaviour is transformed into a parallel behaviour. We consider the Kahn process networks (KPN) to model the parallelized behaviours. KPNs are commonly used for representing parallel behaviours in multimedia and signal processing domains. In this chapter, our objective is to ensure that the generated KPN behaviour is functionally equivalent to the corresponding original sequential behaviour. The parallel process network model, obtained from the sequential behaviour, may again again have be transformed to map it optimally in the available multiprocessor platform. The transformations in this phase affect the code and the concurrency in the process network model. Accordingly, our next objective is to show the equivalence between two KPN behaviours. Also, the deadlock may be introduced in the generated KPNs by both these transformation processes. Our another objective is to detect deadlock in the KPNs.

The chapter is organized as follows: Our overall verification approach is discussed in section 6.2. The KPN model is also introduced in this section. The KPN to ADDG transformation scheme is presented in section 6.3. The correctness of the scheme is developed given in this section. In section 6.4, we show how ADDG based modelling of KPNs helps us detect deadlocks in KPNs. Verification of several KPN level transformations using our technique is discussed in section 6.5. Some experimental results are presented in section 6.6. The contributions of this chapter are summarized in section 6.7.

6.2 Verification framework

As discussed in the introduction section, we consider KPNs as a model of parallel behaviours. Let us briefly introduced the KPN model. We then discuss our approach for verification.

6.2.1 Kahn process networks

Kahn Process Network (KPN) (Kahn, 1974) is a deterministic model of computation which consists of a set of processes and a set of FIFO communication channels. The processes, which are sequential programs, run in parallel. They communicate data among each other through one directional FIFO channels. The FIFOs, which theoretically may be infinite in size, are used for point to point communication among the processes. A process can write any amount of data to a FIFO; it, however, gets blocked when it tries to read from an empty channel. Therefore, the processes of a KPN are synchronized by blocking read property of FIFO. At any given time, a process is either computing or waiting on one of its empty input FIFO channels. FIFOs are represented by data streams and processes by functions that map streams into streams. A stream is a finite or infinite sequence of data elements.

6.2.2 Verification approach

Application programs in signal processing and multimedia domains primarily involve nested loops. The translation process explores the loop level parallelism in such programs. Therefore, processes in the generated KPN also involve nested loops. Since, all the loop bounds are finite, the amount of data communicated through a FIFO channel is also finite. In the original definition of KPN, each process, however, maps a set of input sequences (from the incoming FIFOs) into a set of output sequences (to the outgoing FIFOs) where the input sequences can be finite or infinite. Therefore, a restricted class of KPNs are considered in this work where each process of the KPN is a nested loop program and the amount of data communicated through a FIFO is finite.

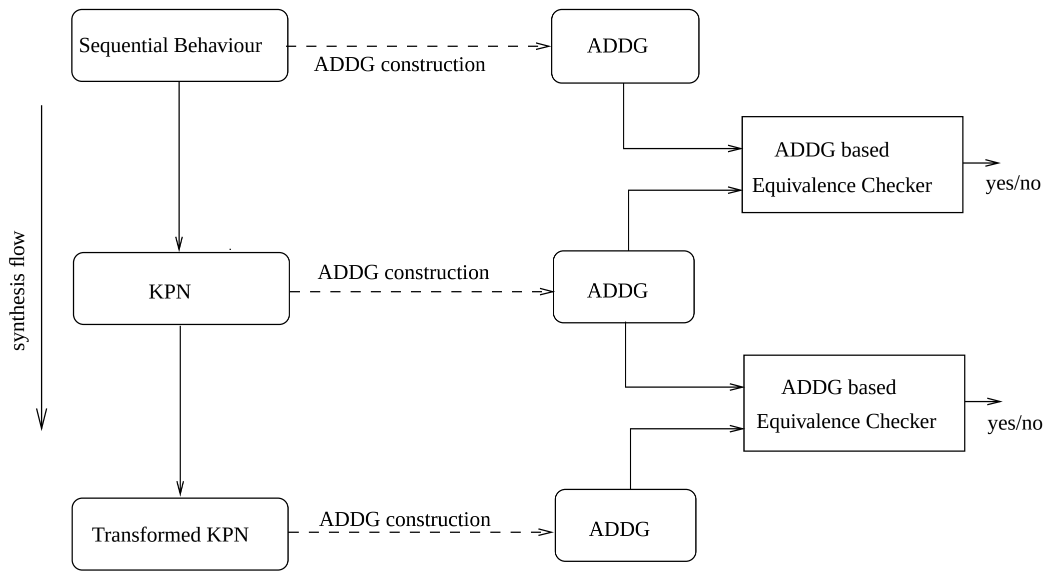


Figure 6.1: Verification of parallelizing transformations by equivalence checking

We leverage the ADDG based equivalence checking developed in the previous chapter for determining equivalence of the sequential behaviour to the derived KPN behaviour and also checking equivalence between a KPN behaviour and its transformed version. Our verification flow is depicted in figure 6.1. Specifically, we first construct array data dependence graphs (ADDGs) from both sequential and KPN behaviours. The equivalence between the sequential behaviour and its corresponding KPN behaviour is then established by checking equivalence between two ADDGs. The mechanism to model a sequential behaviour as an ADDG is already discussed in section 5.2.4. The ADDG based equivalence checking method is also developed in chapter 5. In this chapter, we describe a mechanism to represent a KPN behaviour comprising inherently parallel processes as an ADDG. Once both the behaviours are modelled as ADDGs, their equivalence can be established using the ADDG based equivalence checker.

The equivalence between two KPNs are established by modelling both the KPNs as ADDGs and then applying our ADDG based equivalence checking method. We show in this chapter that our verification framework can handle most of the KPN level transformations.

An example of sequential to KPN code transformation (taken from (Kienhuis et al., 2000)) is given next which serves as a running example for illustrating our method.

Example 21 Let us consider the following nested loop sequential behaviour. It first computes the elements of the array $r1$ from values of two input arrays namely, $in1$ and $in2$. In the next loop, the program computes the values of array $r2$ and finally produces the output array $out1$ from $r2$. The ADDG of the behaviour is shown in figure 5.1(a).

```
for ($i = 1;\ i \leq M,\ i+ = 1$) do
    for ($j = 4;\ j \leq N;\ j+ = 1$) do
        S1: $r1[i+1][j-3] = g_1(in1[i][j],\ in2[i][j])$;
    end for
end for
for ($l = 3;\ l \leq M,\ l+ = 1$) do
    for ($m = 3;\ m \leq N-1;\ m+ = 1$) do
        if ($l+m\ \leq\ 7$) then
            S2: $r2[l][m] = g_2(r1[l-1][m-2])$;
        end if
        if ($l+m\ \geq\ 8$) then
            S3: $r2[l][m] = g_3(r1[l][N-3])$;
        end if
    end for
end for
for ($l = 3;\ l \leq M,\ l+ = 1$) do
    for ($m = 3;\ m \leq N-1;\ m+ = 1$) do
        S4: $out1[l][m] = g_4(r2[l][m])$;
    end for

end for
```

The structure of the KPN behaviour generated by the Compaan tool (Turjan et al., 2004) from the above sequential behaviour is given in figure 6.2. This KPN behaviour consists of four processes and four FIFO channels as shown in the figure. The detailed behaviour of the processes of the KPN are given in figure 6.3. In particular, process $P1$ computes the elements of $r1$, process $P2$ computes those elements of $r2$ which are actually computed by the statement $S2$ in the sequential behaviour, process $P3$ computes those elements of $r2$ which are actually computed by the statement $S3$ in the sequential behaviour and process $P4$ computes the elements of the array $out1$ as in statement $S4$ in the sequential behaviour. Since the elements of $r2$ gets defined by the elements of $r1$, $P1$ sends the required elements of $r1$ to processes $P2$ and $P3$ accordingly through $FIFO1$ and $FIFO2$, respectively. Similarly, the elements of $r2$

are communicated to $P4$ for $out1$ from $P2$ and $P3$ through $FIFO3$ and $FIFO4$, respectively. Therefore, the elements of the arrays $r1$, $r2$, $out1$ are computed in parallel in the KPN. □

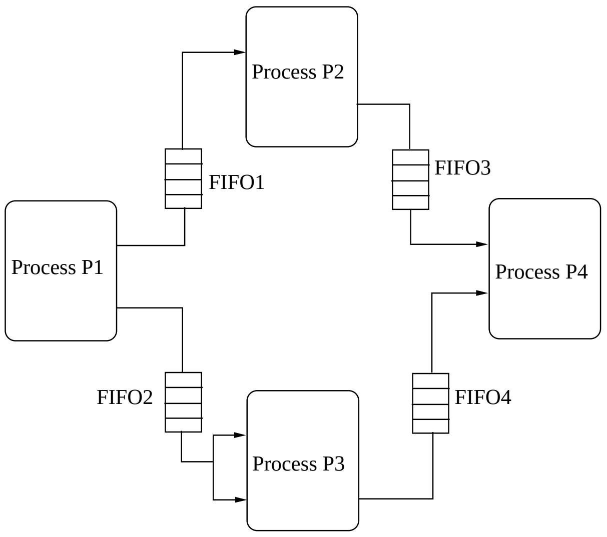


Figure 6.2: The KPN representation of the concurrent behaviour

6.3 Modelling a KPN as an ADDG

The basic steps involved in modelling a KPN as an ADDG are as follows:

1. Each process of the KPN is first modelled as an ADDG.
2. The ADDGs corresponding to the processes are then composed to obtain the ADDG of the KPN.

6.3.1 Modelling KPN processes as ADDGs

As discussed in subsection 6.2.1, a KPN consists of a set of sequential processes which execute in parallel. They communicate via a set of point to point, one way FIFOs. The processes are synchronized by way of blocking read property of the FIFOs. Each process of a KPN behaviour is sequential. It is easily seen that each individual process in

Behaviour of Process P1:

```
for ($i = 1;\ i \leq M;\ i+ = 1$) do
    for ($j = 4;\ j \leq N;\ j+ = 1$) do
        if ($-i - j + 6 \geq 0$ and $-i + M - 2 \geq 0$) then
            $FIFO1.put(g_1(in1[i][j],\ in2[i][j]))$;
        end if
        if ($j - N == 0$ and $i + M - 8 \geq 0$ and $-i + M - 1 \geq 0$) then
            $FIFO2.put(g_1(in1[i][j],\ in2[i][j]))$;
        end if
    end for

end for
```

Behaviour of Process P2:

```
for ($l = 3;\ l \leq M,\ l+ = 1$) do
    for ($m = 3;\ m \leq N - 1;\ m+ = 1$) do
        if ($l - 4 \leq 0$ and $m + l - 7 \leq 0$) then
            $t1 = FIFO1.get()$;
        end if
        if ($l - 4 \leq 0$ and $m + l - 7 \leq 0$) then
            $FIFO3.put(g_2(t1))$;
        end if
    end for

end for
```

Behaviour of Process P3:

```
for ($l = 3;\ l \leq M,\ l+ = 1$) do
    for ($m = 3;\ m \leq N - 1;\ m+ = 1$) do
        if ($9 - N \leq l$ and $8 - l \leq m$ and $l + m - 8 == 0$ and $5 - m \geq 0$ and $m + M - 8 \geq 0$) then
            $t2 = FIFO2.get()$;
        else if ($9 - N \leq l$ and $8 - l \leq m$ and $m - 3 == 0$ and $l - 6 \geq 0$) then
            $t2 = FIFO2.get()$;
        end if
        if ($9 - N \leq 0$ and $8 - l \leq m$) then
            $FIFO4.put(g_3(t2))$;
        end if
    end for

end for
```

Behaviour of Process P4:

```
for ($l = 3;\ l \leq M,\ l+ = 1$) do
    for ($m = 3;\ m \leq N - 1;\ m+ = 1$) do
        if ($l + m \leq 7$) then
            $t3 = FIFO3.get()$;
        end if
        if ($l + m \geq 8$) then
            $t3 = FIFO4.get()$;
        end if
        $out[l-3][m-3] = g_4(t3))$;
    end for

end for
```

Figure 6.3: Behaviours of the processes of the KPN given in figure 6.2

the KPN behaviour except for the *get* and the *put* FIFO primitives within the processes is otherwise a sequential program using arrays and can be modelled using ADDGs. We shall show that these two primitives can also be encompassed within the ADDG framework by utilizing the first-in-first-out property of FIFOs. We may model a FIFO channel as a one dimensional array associated with two indices, one each for handling the primitive operations of *get* and *put*, respectively for reading and writing elements from the start of the array one by one. Let the index corresponding to a *get* operation be denoted as ‘out-pointer’ and that for a *put* operation be denoted as ‘in-pointer’. Let the FIFO F communicate data from a process P_1 to another process P_2 in a KPN. In the ADDG, the FIFO F is realized as a one dimensional array F with in-pointer FIn and out-pointer $FOut$, say. The statement $F.put(x)$ in P_1 is visualized as the assignment $F[++FIn] = x$ in P_1. Similarly, the statement $y = F.get()$ in P_2 is now visualized as $y = F[FOut++]$ in P_2. The modified behaviours of the KPN processes given in figure 6.3 are shown in figure 6.4. The individual variables are converted to arrays in order

Behaviour of Process P1:

```
for ($i = 1, f1In = -1, f2In = -1;\ i \le M,\ i+ = 1$) do
    for ($j = 4;\ j \le N;\ j+ = 1$) do
        if ($-i - j + 6 \ge 0 \wedge -i + M - 2 \ge 0$) then
            S11: $FIFO1[++f1In] = g_1(in1[i][j],\ in2[i][j])$;
        end if
        if ($j - N == 0 \wedge i + M - 8 \ge 0 \wedge -i + M - 1 \ge 0$)
        then
            S12: $FIFO2[++f2In] = g_1(in1[i][j],\ in2[i][j])$;
        end if
    end for

end for
```

Behaviour of Process P2:

```
for ($l = 3, f1Out = -1, f3In = -1;\ l \le M,\ l+ = 1$) do
    for ($m = 3;\ m \le N - 1;\ m+ = 1$) do
        if ($l - 4 \le 0 \wedge m + l - 7 \le 0$) then
            S13: $t1[l][m] = FIFO1[++f1Out]$;
        else
            S14: $t1[l][m] = FIFO1[f1Out]$;
        end if
        if ($l - 4 \le 0 \wedge m + l - 7 \le 0$) then
            S15: $FIFO3[++f3In] = g_2(t1[l][m])$;
        end if
    end for

end for
```

Behaviour of Process P3:

```
for ($l = 3, f2Out = -1;\ l \le M,\ l+ = 1$) do
    for ($m = 3;\ m \le N - 1;\ m+ = 1$) do
        if ($9 - N \le l$ and $8 - l \le m$ and $l + m - 8 == 0$ and
        $5 - m \ge 0$ and $m + M - 8 \ge 0$) then
            S16: $t2[l][m] = FIFO2[++f2Out]$;
        else if ($9 - N \le l$ and $8 - l \le m$ and $m - 3 == 0$
        and $l - 6 \ge 0$) then
            S17: $t2[l][m] = FIFO2[++f2Out]$;
        else
            S18: $t2[l][m] = FIFO2[f2Out]$;
        end if
        if ($9 - N \le 0$ and $8 - l \le m$) then
            S19: $FIFO4[++f4In] = (g_3(t2[l][m])$;
        end if
    end for

end for
```

Behaviour of Process P4:

```
for ($l = 3, f3Out = -1, f4Out = -1;\ l \le M,\ l+ = 1$) do
    for ($m = 3;\ m \le N - 1;\ m+ = 1$) do
        if ($l + m \le 7$) then
            S20: $t3[l][m] = FIFO3[++f3Out]$;
        end if
        if ($l + m \ge 8$) then
            S21: $t3[l][m] = FIFO4[++f3Out]$;
        end if
        S22: $out[l - 3][m - 3] = g_4(t3[l][m])$;
    end for

end for
```

Figure 6.4: Modified behaviours of the processes given in figure 6.3

to represent them as ADDGs in the above behaviour.

Each process of the KPN can be modelled as an ADDG when its FIFOs are realized as arrays as identified above. The ADDGs of the four processes of the KPN in figure 6.4 are depicted in figure 6.5. It may be noted that the array corresponding to a FIFO becomes a start node in the producer process and a terminal node in the consumer process. The FIFO $FIFO1$, for example, is communicating from the producer process $P1$ to the consumer process $P2$ in the KPN of figure 6.4. It may be observed that $FIFO1$ is a start node in the ADDG corresponding to $P1$ and a terminal node in the ADDG corresponding to P_2 in figures 6.5(a) and 6.5(b), respectively. We refer to a FIFO and the linear array corresponding to it by the same name.

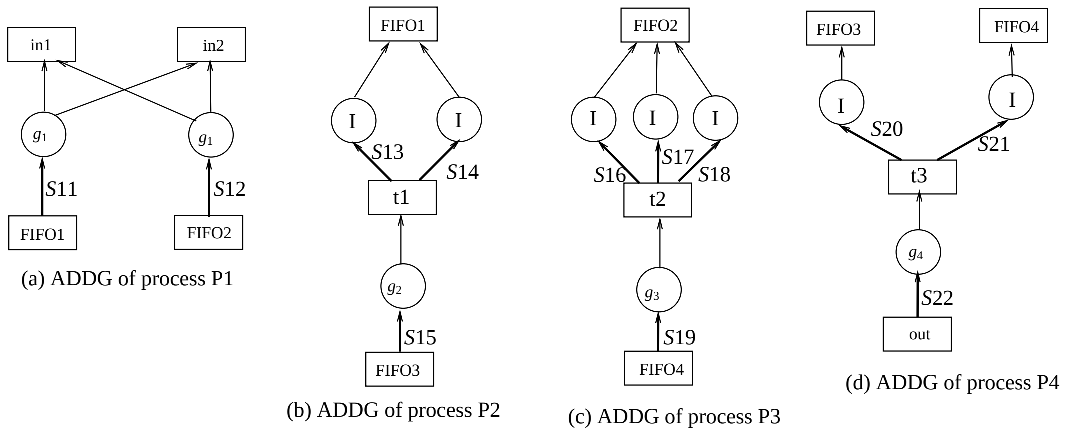


Figure 6.5: The ADDGs of processes of the KPN behaviour in figure 6.2

6.3.2 Computation of the dependence mappings involving FIFOs

There is still one problem in encoding these modified sequential behaviours by ADDGs. The elements of the linear array is either computed (in a producer process) from the values of other operand arrays or used (in a consumer process) to compute the element values of other arrays under a nested loop structure. For the former case, the definition mapping and its domain, and for the later case, the operand mapping and its domain, of the linear array need to be computed for the dependency mapping between the defined array and the used array in the statement of concern. The exact mapping between the indices of the loop under which the statement is executed and the index of the linear array, however, is missing because the index expressions (basically a variable) of the linear array are not in terms of the loop indices. So, we need to represent the index variable of the linear array (corresponding to each FIFO) in terms of the loop indices under which the statement is executed. In figure 6.5(a), for instance, the definition mapping and its domain for the linear array $FIFO1$ are required to obtain the dependency mappings ${}_{S11}M_{FIFO1,\, in1}$ and ${}_{S11}M_{FIFO1,\, in2}$ between the indices of the arrays $in1$ and $in2$ and the index $f1In$ of the array $FIFO1$. For this, we need to express $f1In$ in terms of the loop indices i and j.

Let us consider a general behaviour given in figure 6.6 to evolve a formulation of this mapping. In this behaviour, the array FIFO1 is computed from the values of m multi-dimensional arrays $u_1, \ldots, u_m$ in a loop with a nesting depth x. Here, $e_{i1},\ \ldots,\ e_{il_i}, 1 \le i \le m$, are affine expressions and C_D is some condition over the loop

```
$for(f1In = 0, i_1 = L_1;\ i_1 \le H_1;\ i_1 += r_1)$
  $for(i_2 = L_2;\ i_2 \le H_2;\ i_2 += r_2)$
    $\vdots$
      $for(i_x = L_x; i_x \le H_x; i_x += r_x)$
        $if(C_D)\ then$
          $P:\ FIFO1[f1In{+}{+}] = F(u_1[e_{11}]\ldots[e_{1l_1}],\ \ldots,\ u_m[e_{m1}]\ldots[e_{ml_m}]);$
```

Figure 6.6: A generalized nested loop behaviour

indices $i_1, \ldots, i_x$. Our objective is to find the relation between $f1In$ and $i_1, \ldots, i_x$. This relation may be linear or non-linear. It can be mechanically found out whether the relation is linear or not. In the following we evolve the mechanical steps of synthesizing the relation through a case analysis.

Case 1: The relation is linear: We have the following sub-cases:

Subcase 1.1: The condition C_D is identically true:
Here the closed form of the linear relation is given by the following equation:

$$\begin{aligned} f1In = {} & (i_1 - L_1)/r_1 \times \{((H_2 - L_2)/r_2 + 1)\ \times\ \ldots\ \times\ ((H_x - L_x)/r_x + 1)\} \\ & + (i_2 - L_2)/r_2 \times \{((H_3 - L_3)/r_3 + 1)\ \times\ \ldots\ \times\ ((H_x - L_x)/r_x + 1)\} \\ & + \ldots + (i_x - L_x)/r_x \end{aligned} \tag{6.1}$$

Subcase 1.2: The condition C_D is not identically true: The relation may or may not involve all the loop indices $i_1, \ldots, i_x$. Accordingly, we have two subcases.

Subcase 1.2.1: The relation involves all the loop index variables:
The general form of the relation between $f1In$ and $i_1, \ldots, i_x$ is

$$f1In = i_1 \times \alpha_1 + i_2 \times \alpha_2 + \ldots + i_x \times \alpha_x + \alpha_{x+1}, \tag{6.2}$$

where $\alpha_1, \ldots, \alpha_{x+1}$ are real constants whose values are to be found. Let in a particular iteration l, the value of i_k be $a_{l,k}$, $1 \le k \le x$, and the value of $f1In$ be c_l. Placing these values in equation 6.2, we have

$$a_{l,1} \times \alpha_1 \ + \ a_{l,2} \times \alpha_2 \ + \ \ldots \ + \ a_{l,x} \times \alpha_x \ + \ \alpha_{x+1} = c_l \tag{6.3}$$

Here, $a_{l,1}, \ \ldots, \ a_{l,x}, \ c_l$, are some known integer values obtained through exhaustive simulation of the loop from the first to the l^{th} iterations. Let n iterations of the loop body satisfy the condition C_D. Considering sequentially all such n iterations of the loop, we can have n equations of the form

$$\begin{aligned}
a_{1,1} \times \alpha_1 \ + \ a_{1,2} \times \alpha_2 \ + \ \ldots \ + \ a_{1,x} \times \alpha_x \ + \ \alpha_{x+1} = c_1 \\
a_{2,1} \times \alpha_1 \ + \ a_{2,2} \times \alpha_2 \ + \ \ldots \ + \ a_{2,x} \times \alpha_x \ + \ \alpha_{x+1} = c_2 \\
\vdots \\
a_{n,1} \times \alpha_1 \ + \ a_{n,2} \times \alpha_2 \ + \ \ldots \ + \ a_{n,x} \times \alpha_x \ + \ \alpha_{x+1} = c_n
\end{aligned}$$

Solving these equations by any linear program solvers such as, lp_solve (lp_solve, 2010) or any SMT solver such as, Yices (Yices, 2010), we can obtain the values of $\alpha_1, \ \ldots, \ \alpha_{x+1}$ and hence the relation between $f1In$ and $i_1, \ \ldots, \ i_x$. Following example illustrates the above mechanism.

Example 22 Let us consider the statement S11 of $P1$ in the bebaviour given in figure 6.4. Let us now extract the relation between the index $f1In$ of $FIFO1$ and the loop indices i and j. The relation can be represented as

$$f1In = i \times \alpha_1 + j \times \alpha_2 + \alpha_3$$

We need to find the values of α_1 α_2 and α_3. Let us consider the first iteration $i = 1, j = 4$ with $f1In = 0$, we have

$$\alpha_1 + 4 \times \alpha_2 + \alpha_3 = 0$$

For the second iteration $i = 1, j = 5$ with $f1In = 1$, we have

$$\alpha_1 + 5 \times \alpha_2 + \alpha_3 = 1$$

The statement $S11$ does not execute for the remaining iterations with $i = 1$ (and $6 \leq j \leq M$) since the condition C_D (i.e., $-i - j + 6 \geq 0 \wedge -i + M - 2 \geq 0$) under which S11 executes fails in these iterations. For the next iteration which satisfies C_D, $i = 2, j = 4$ with $f1In = 2$; we have

$$2 \times \alpha_1 + 4 \times \alpha_2 + \alpha_3 = 2$$

None of the remaining iterations of the loop satisfies the condition $-i - j + 6 \geq 0$ $\wedge - i + M - 2 \geq 0$. Therefore, the statement S1 executes only three times. Solving the above three equations, we have $\alpha_1 = 2$, $\alpha_2 = 1$, $\alpha_3 = -6$. Therefore, $f1In = 2 \times i + j - 6$. □

The range $(H_k - L_k + 1)$ of the loop index i_k, $1 \leq k \leq x$, can be high in practice. Consequently, the number of iterations that satisfy C_D may also be large. Hence, the number of equations given to the solver is quite high. Since, we have $x + 1$ unknown variables in those equations, only $x + 1$ linearly independent equations actually suffice to find the values of $\alpha_1, \ldots, \alpha_{x+1}$. Unfortunately, it is not possible to choose such $x + 1$ linearly independent equations greedily from n equations avoiding exhaustive simulation. The following example illustrates that if we stop after obtaining the first $x + 1$ linearly independent equations, then we reach a linear relation wrongly (because the relation is actually non linear).

```
for(i=1, fIn=-1; i < 6; i++)
 for(j=1; j < 6; j++)
  if(j <= i)
    S1: FIFO1[++f1In] = a[i][j];
```

(a) behaviour of a process

	5					14
	4				9	13
	3			5	8	12
↑j	2		2	4	7	11
	1	0	1	3	6	10
		1	2	3	4	5
				$\overrightarrow{i}$		

(b) relation between f1In and (i,j)

Figure 6.7: An example of nonlinear relation between $f1In$ and loop indices (i, j)

Example 23 Let us consider the behaviour of the process given in figure 6.7(a). The corresponding values of $f1In$ for different values of (i, j) of this behaviour is shown in figure 6.7(b). Here, the relation can be written as $f1In = i \times \alpha_1 + j \times \alpha_2 + \alpha_3$. Since we have three unknown variables here, three linear independent equations should suffice to find them. Considering the first three iterations of the loop, we have

$\alpha_1 + \alpha_2 + \alpha_3 = 0,$

$2 \times \alpha_1 + \alpha_2 + \alpha_3 = 1$ and

$2 \times \alpha_1 + 2 \times \alpha_2 + \alpha_3 = 2.$

Solving these three equations, we find that $\alpha_1 = 1, \alpha_2 = 1, \alpha_3 = -1$. So the synthesized relation is $f1In = i + j - 2$. Now, consider the iteration with $i = 4, j = 3$, the value of $f1In$ by this equation is 5. However, the table in figure 6.7(b) suggests that the actual value of $f1In$ is 8 for this iteration. Therefore, inferring the relation from the first three equations, which are linearly independent, is wrong. We shall show in example 26 that the relation is actually non-liner for this behaviour. $\square$

It is, however, possible to prune out some of the input equations to save the running time of lp_solve or Yices for the following case. If the iteration vectors $\vec{v_1}, \ldots, \vec{v_n}$ are multiples of another iteration vector $\vec{v_p}$ and subtracting the equation obtained by $\vec{v_p}$ from the equations obtained by vectors $\vec{v_1}, \ldots, \vec{v_n}$ result in a single equation, then we can ignore the equations corresponding to $\vec{v_1}, \ldots, \vec{v_n}$ for solving the whole system of equations. It may be noted that we still need an exhaustive simulation of the entire loop since we need to find all the equations; the above step only helps in reducing the size of the input to the lp_solve or Yices. The following example illustrates the fact.

```
for(i=0; i<=20; i+=1)
   if(i % 2 == 0){
      S1: a[i] = FIFO1.get();
   c[i] = f1(a[i]); }
```

Figure 6.8: An process of a KPN

Example 24 Let us consider the process in figure 6.8. The relation between out-pointer $FIFO1Out$ of $FIFO1$ and the loop index i can be represented as $FIFO1Out = \alpha_1 \times i + \alpha_2$. It may be noted that the statement $S1$ executes eleven times. So, we can obtain eleven equations for $i = 0, 2, \ldots, 20$, respectively, namely $0 \times \alpha_1 + \alpha_2 = 0$, $2 \times \alpha_1 + \alpha_2 = 1$, $4 \times \alpha_1 + \alpha_2 = 2, \ldots, 20 \times \alpha_1 + \alpha_2 = 10$. As discussed above, the values of α_1 and α_2 can be obtained by solving all these equations using lp_solve or

Yices. Obviously, the first two equations suffice to obtain (using lp_solve or Yices) the relation $FIFO1Out = i/2$ leading to the dependence mapping $_{S1}M_{a,FIFO2} = \{[i] \rightarrow [i/2] \mid 0 \leq i \leq 10 \wedge \exists p \in \mathbb{N}(i = 2 \times p)\}$. Therefore, we can reduce the time taken by the solver by removing most of the redundant equations from the inputs of the solver. It may be noted that the values of $i = 2, 4, \ldots, 20$ are multiple of the value of $i = 1$. If the equation obtained for $i = 1$ is subtracted from the equations obtained for $i = 2, \ldots, 20$, then we shall obtain nine equations of the form $2k \times \alpha_1 = k,\ 1 \leq k \leq 9$. All these nine equations reduce to a single equation of the form $2 \times \alpha_1 = 1$. Therefore, we select two equations $\alpha_1 \times 0 + \alpha_2 = 0$ and $\alpha \times 2 + \alpha_2 = 1$ from the above eleven equations and submit to lp_solve or Yices. □

Subcase 1.2.2: The relation is independent of some loop index variables:
In this case, the linear system of equations is under constrained, i.e., among the equations obtained through exhaustive enumeration of the loops, there is actually less than $x+1$ linearly independent equations. For an under constrained linear system, there are infinite solutions, i.e., an infinite number of assignments to $\alpha_1, \ldots, \alpha_{x+1}$ satisfy those equations. The value $f1In$ ($f1Out$) remains the same for all the assignments. The solver returns one of the possible solutions. The following example illustrates this subcase.

```
for(i=0, fIn=-1; i<=10; i+=1)
   for(j=0; j<=10; j++)
      if(i == j)
         S1: FIFO[++f1In] = f1(a[i]);
```

Figure 6.9: An example for under constrained linear system of equations

Example 25 Let us consider the behaviour of the producer process given in figure 6.9. Clearly, the statement $S1$ iterates 11 times. So, we find 11 equations of the form $k \times \alpha_1 + k \times \alpha_2 + \alpha_3 = k,\ 0 \leq k \leq 10$. For $k = 0$, we obtain $\alpha_3 = 0$. By applying the equation reduction strategy described above, the remaining 10 equations, which are linearly dependent, reduce to $\alpha_1 + \alpha_2 = 1$. No unique value of α_1 and α_2 cannot be obtained since we have only one equation over them. Here, we have an infinite number of assignments to α_1 and α_2 of the form $\alpha_1 = x$ and $\alpha_2 = -(x-1),\ \forall x \in \mathbb{N}$. The solver returns any one of them as a solution.

It will be observed subsequently that computation of transitive dependence over a FIFO necessiates equating $f1In$ equal to $f1Out$. If the solution returned by the solver for $f1In$ differs from that returned for $f1Out$ (for the under constrained cases), the process is not impaired this process (because equating $f1In = f1Out$ takes place under the explicit condition C_D). In particular, for this example, let us assume that we also have the same loop in the consumer process where data from $FIFO1$ are read. Let the solver returns $\alpha_1 = p$ and $\alpha_2 = -(p-1),\ p \in \mathbb{N}$ for fIn and returns $\alpha_1 = q$ and $\alpha_2 = -(q-1),\ q \in \mathbb{N}$ for $fOut$. Based on the values returned by the solver, we obtain $fIn = p \times i - (p-1) \times j$ and $fOut = q \times i - (q-1) \times j$. Clearly $fIn = fOut, \forall i, j,$ $0 \leq i \leq 10 \wedge 0 \leq j \leq 10 \wedge i == j$. Therefore, an arbitrary choice does not hamper the dependence mapping computations over a FIFO. □

Case 2: The relation is not linear:
In this case, there does not exist any assignment of $\alpha_1, \ \ldots, \ \alpha_{x+1}$ which satisfy all the above equations. The solver reports that the constraints are unsatisfiable. The following example illustrates the fact.

Example 26 Let us revisit the behaviour of the process in figure 6.7(a). The corresponding values of $f1In$ for different values of (i, j) of this behaviour is shown in figure 6.7(b). It can be shown that the relation between $f1In$ and i, j is $f1In = (i \times (i-1))/2 + (j-1)$ (and the corresponding dependence mapping is

$_{S1}M_{FIFO,a} = \{[(i \times (i-1))/2 + (j-1)] \rightarrow [i][j] \mid 0 \leq i \leq 5 \wedge 0 \leq j \leq 5\}$). Clearly this is a nonlinear relation. In this case our method will generate 15 equations (corresponding to 15 entries of the table 6.7(b)). When these equations are submitted to lp_solve or Yices, the latter reports that the constraints are unsatisfiable indicating, thereby,that no linear relation exists. □

As discussed in the previous chapter (in section 5.8), our ADDG equivalence checking method uses Omega calculator (Kelly et al., 2008) for computing the dependence mappings. The inherent limitation of Omega calculator is that it only supports Presburger formulas[1]. Therefore, such nonlinear dependence mappings cannot be handled by Omega calculator.

[1] Presburger formulas contain affine constraints, the usual logical connectives, and existential and universal quantifiers

The closed form of the relation between $f1In$ and $i_1,\ \ldots,\ i_x$ will be elusive for nonlinear cases. In such cases, the dependence mapping between $FIFO1$ and any right hand side (rhs) array $u_n,\ 1 \le n \le m$, in the statement P of figure 6.6 can still be synthesized as follows:

The iteration domain of the statement P in figure 6.6 is:

$$I_P = \{[i_1, i_2, \ldots, i_x] \mid \bigwedge_{k=1}^{x} (L_k \le i_k \le H_k \wedge C_D \wedge \exists \alpha_k \in \mathbb{N}(i_k = \alpha_k r_k + L_k))\}$$

The definition domain of FIFO1 in the statement P is

$${}_PD_{FIFO1} = \{i \mid 0 \le i \le (H_1 - L_1 + 1) \times\ \ldots\ (H_x - L_x + 1)\}.$$

The operand domain of the array u_n is

$${}_PU_{u_n} \subseteq \mathbb{Z}^{l_n} = \{[e_{n1}(\vec{v}),\ \ldots,\ e_{nl_n}(\vec{v})] \mid \vec{v} \in I_P\}.$$

The dependence mapping can be represented as

$$\begin{aligned} {}_PM_{FIFO1,u_n} = \ & \{\ [f1In] \to [\vec{v}] \mid f1In \in\ {}_PD_{FIFO1}\ \wedge\ \vec{v} \in\ {}_PU_{u_n} \\ & \wedge\ \forall \vec{v}_1, \vec{v}_2 \in\ {}_PU_{u_n}(\vec{v}_1 \le \vec{v}_2\ \wedge \nexists \vec{v}_3 \in\ {}_PU_{u_n}(\vec{v}_1 \le \vec{v}_3\ \wedge\ \vec{v}_3 \le \vec{v}_2)) \\ & \implies\ \exists f1In_1, f1In_2 \in\ {}_PD_{FIFO1}((([f1In_1] \to [\vec{v}_1]) \in\ {}_PM_{FIFO1,u_n} \\ & \wedge ([f1In_2] \to [\vec{v}_2]) \in\ {}_PM_{FIFO1,u_n}\ \wedge\ (f1In_1 = f1In_2 - 1))\}, \\ & where\ \vec{v}_i \le \vec{v}_j,\ 1 \le i, j \le 3,\ denotes\ the\ lexicographic\ ordering \\ & of\ the\ vectors\ \vec{v}_i\ and\ \vec{v}_j \end{aligned} \tag{6.4}$$

The expression depicted in equation 6.4 is arrived at by taking two consecutive iteration vectors which satisfy C_D and relating them to the corresponding $f1In$ values $f1In1$ and $f1In2$, respectively. Unfortunately, Omega calculator does not work on such inductive definition of the dependence mapping. Therefore, when the relation between $f1In$ and $i_1,\ \ldots,\ i_x$ is not linear, we have to put the relations in their exhaustively enumerated forms to the Omega calculator. Accordingly, the dependence mapping for the statement $S1$ in figure 6.7(a) is represented as

$$_{S1}M_{FIFO1,a} = \{[0] \rightarrow [1][1]\} \cup \{[1] \rightarrow [2][1]\} \cup \{[2] \rightarrow [2][2]\} \cup \{[3] \rightarrow [3][1]\} \\ \cup \ \ldots \ \cup \{[13] \rightarrow [5][4]\} \cup \{[14] \rightarrow [5][5]\}$$

A FIFO written by multiple statements of a process

So far we have discussed a procedure to obtain the relation between the FIFO pointer and the loop indices through case analysis considering that only one statement of a producer process produces data elements into a FIFO. In general, however, more than one statement of a producer process may put data into the same FIFO. Similarly, there may be more than one statement of a consumer process consuming data from a FIFO. In such cases, we need the starting value of the FIFO pointers for each of such statements of a process. Following example illustrates the phenomenon.

```
for(i=0, fIn=-1; i<10; i+=1)
   for(j=0; j<10; j++){
       S1: FIFO[++f1In] = f1(a[i]);
       S2: FIFO[++f1In] = f2(a[i]); }
for(k=0; k<=10; k+=1)
   S3: FIFO[++f1In] = f3(a[k]);
```

Figure 6.10: A FIFO is written by more than one statement

Example 27 Let us consider the producer process given in figure 6.10 where the elements of $FIFO1$ are written by three statements. For the statements $S1$ and $S2$, the starting values of $f1In$ are 0 and 1, respectively. It may be noted that the statements $S1$ and $S2$ of the first two dimensional loop write 0 to 199 locations of the $FIFO1$ alternately. Therefore, the starting value of $f1In$ for the second loop is 200. This value will be used to find the relation between $f1In$ and the loop index k of the second loop. □

The starting value of a FIFO pointer for a statement (in a loop) can be obtained by completely simulating the loop bodies where the previous elements of FIFO are written. However, if the condition C_D for a loop is identically true, then the initial values of $f1In$ for the statements can be obtained directly.

In general, there may be n such statements within a loop writing data into a FIFO. As discussed in case 1.1, if there is a single statement within a loop body and C_D is true, the relation between $f1In$ and the loop indices can be obtained by equation 6.1. If there are n statements in a loop writing data into a FIFO, then the relation $f1In$ and the loop indices for the m^{th} statement, $1 \leq m \leq n$, can by obtained with the following modification of equation 6.1:

$$\begin{aligned} f1In = \ & [(i_1 - L_1)/r_1 \times \{((H_2 - L_2)/r_2 + 1) \ \times \ \ldots \ \times \ ((H_x - L_x)/r_x + 1)\} \\ & + \ (i_2 - L_2)/r_2 \times \{((H_3 - L_3)/r_3 + 1) \ \times \ \ldots \ \times \ ((H_x - L_x)/r_x + 1)\} \\ & + \ \ldots \ + \ (i_x - L_x)/r_x] \times m + f1In_0, \end{aligned} \tag{6.5}$$

where $f1In_0$ is the starting value of $f1In$ for the m^{th} statement. The final value of the n^{th} statement of a loop body can be obtained by substituting respectively $H_1,\ H_2,\ \ldots,\ H_x$ in place of $i_1,\ i_2,\ \ldots,\ i_x$ in the equation 6.5. Thus, to obtain the starting value of $f1In$ for the statement $S3$ in figure 6.10, the final value of $f1In$ for the statement $S2$ is obtained by equation 6.5 as $(9 \times 10 + 9) \times 2 + 1 = 199$. Therefore, the starting value of $f1In$ is 200 for the second loop. The relation between $f1In$ and the index k of the second loop for the statement $S3$ can be obtained using the equation 6.5 with $m = 1$ and $f1In_0 = 200$. Hence, the relation is obtained as computed thereby as $f1In = k + 100$.

Finally, our scheme to obtain the relation between $f1In$ ($f1Out$) and $i_1,\ \ldots,\ i_x$ is given as algorithm 9.

6.3.3 Composition of ADDGs of KPN processes

The next task is to compose the ADDGs obtained from the individual processes to obtain the ADDG of the KPN. The composed ADDG $G = (V,E)$ of two ADDGs $G_1 = (V_1,E_1)$ and $G_2 = (V_2,E_2)$ would be the union of their respective vertices, i.e., $V = V_1 \cup V_2$ and edges, i.e., $E = E_1 \cup E_2$. It may be recalled that a FIFO is used as the communication medium between a producer process and a consumer process in a KPN. Let F_k be the FIFO used as the communication channel between the producer process P_i and the consumer process P_j. Let G_i and G_j be the ADDGs obtained from P_i and P_j, respectively, with F_k depicted as a linear array. In this case, F_k would be a start node in G_i and a terminal node in G_j. Therefore, the composed ADDG of G_i and G_j captures the transitive dependence of the start node of G_j on the terminal nodes of

Algorithm 9 Algorithm to find relation between $f1In$ ($f1Out$) and loop indices

```
/* Input: statement label
Output: the relation */
 1: Obtain the starting value of $f1In$ ($f1Out$).
 2: if all $L_i$ and $H_i$ are integer constants and $C_D$ is true then
 3:     Obtain the relation using the equation 6.5;
 4: else
 5:     Find all the linear equations from the iterations of the loops;
 6:     Remove the redundant equations as described in subcase 1.2.1;
 7:     Apply Yices to solve these linear equations;
 8:     if Yices is able to obtain a linear relation then
 9:         /*Yices yields a solution */
10:         Use the expression returned by Yices;
11:     else
12:         /* Yices indicates that the equations are unsatisfiable */
13:         Use the relation in enumerated form;
14:     end if
15: end if
```

G_i. While computing the transitive dependence across the linear array corresponding to the FIFO channel F_k, we use the ‘first-in-first-out’ property of the FIFO; in other words, we make the in-pointer fIn of F_k equal to the out-pointer $fOut$ of F_k to obtain the transitive dependence. On the other hand, if there is no FIFO communication between two processes, then their ADDGs remain disjoint in the composed ADDG. Therefore, the ADDG corresponding to the KPN can be obtained from the ADDGs of the individual processes by composing them along all the FIFOs of the KPN.

The composition of two ADDGs can be formally defined as follows:

Let $G_1 = (V_1,\ E_1)$ and $G_2 = (V_2, E_2)$ be the respective ADDGs of any two processes P_1 and P_2 of a KPN. Let there be k FIFOs $F_1,\ \ldots,\ F_k$ between P_1 and P_2. We assume that P_1 and P_2 may have some common input and output arrays, but all the internal arrays of the processes have distinct names. Let P_1 and P_2 have m common input arrays $I_1,\ \ldots,\ I_m$ and n common output arrays $O_1,\ \ldots O_n$. Therefore, $V_1 \cap V_2 = \{F_1,\ \ldots,\ F_k,\ I_1,\ \ldots,\ I_m,\ O_1,\ \ldots O_n\}$. The composed ADDG G_{12} of G_1 and G_2 is $G_{12} = G_1 \diamond G_2 = (V_1 \cup V_2,\ E_1 \cup E_2)$. The input array nodes $I_1,\ \ldots,\ I_m$ become the terminal nodes and the output array nodes $O_1,\ \ldots O_n$ become the start nodes in the composed ADDG G_{12}. The linear arrays $F_1,\ \ldots,\ F_k$ are either start nodes or terminal nodes in G_1 and G_2 depending upon the direction of the FIFOs $F_1,\ \ldots,\ F_k$. Let F_1 be a start node in G_1 and a terminal node in G_2. It means that

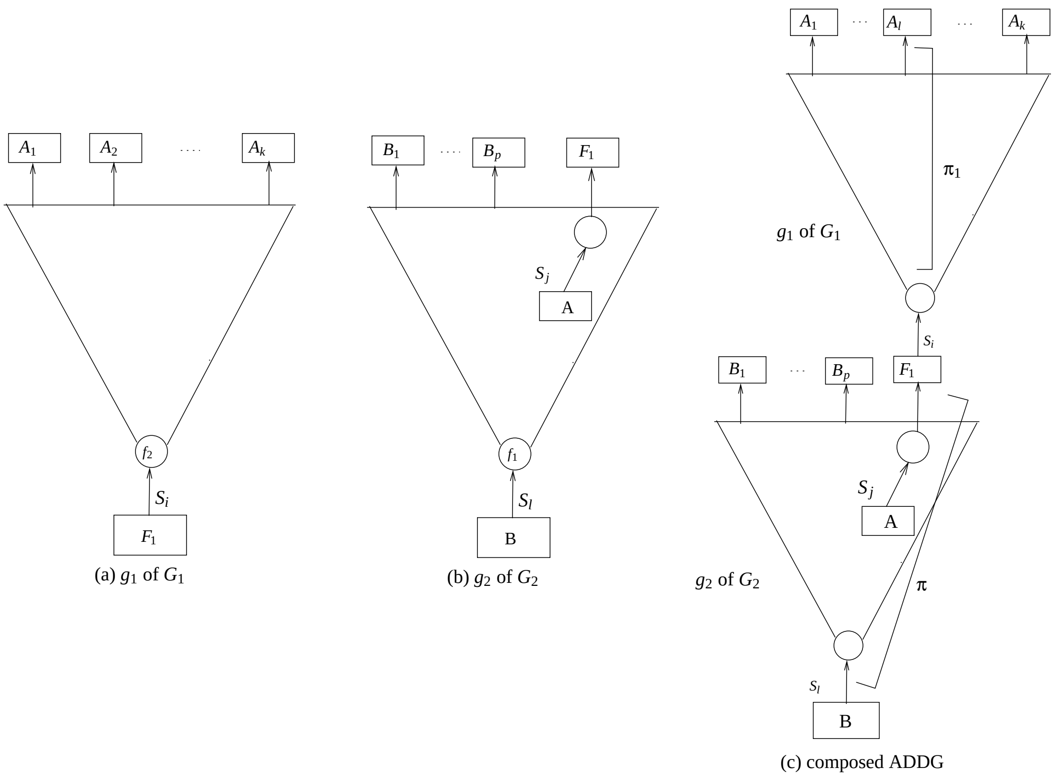


Figure 6.11: Composition of ADDGs

the direction of F_1 is from (producer) process P_1 to (consumer) process P_2. The linear array nodes $F_l,\ 1 \le l \le k$, become internal array node in G_{12}. Specifically, we need to compose the slices in G_2, which have F_1 as terminal nodes, with the slices in G_1, which have F_1 as start nodes, to obtain the transitive dependencies between the output and the input arrays in G_{12}. Let $g_2(B, \langle B_1,\ \ldots,\ B_p, F_1\rangle)$ be one such slice in G_1 and $g_1(F_1, \langle A_1,\ \ldots,\ A_k\rangle)$ be one such slice in G_2. The slices g_1 and g_2 are depicted as figures 6.11(a) and 6.11(b), respectively. Let the characteristic formulas of the two slices g_2 and g_1 be $\tau_{g_2} = \langle f_1(B_1,\ \ldots,\ B_p, F_1),\ \langle {}_{g_2}M_{B,B_1},\ \ldots,\ {}_{g_2}M_{B,B_p},\ {}_{g_2}M_{B,F_1}\rangle\rangle$ and $\tau_{g_1} = \langle f_2(A_1,\ \ldots,\ A_k),\ \langle {}_{g_1}M_{F_1,A_2},\ \ldots,\ {}_{g_1}M_{F_1,A_k}\rangle\rangle$, respectively. Let the slice $g(B, \langle B_1,\ \ldots,\ B_p,\ A_1,\ \ldots,\ A_k\rangle)$ as shown in figure 6.11(c) be the composition of g_1 and g_2, i.e., $g = g_2 \diamond g_1$. The characteristic formula of g is

$$\tau_g = \langle f_1(B_1, \ldots, B_p,\ f_2(A_1,\ \ldots,\ A_k)),\ \langle\ {}_{g_2}M_{B,B_1},\ \ldots, {}_{g_2}M_{B,B_p}, {}_gM_{B,A_1}, \ldots, {}_gM_{B,A_k}\rangle\rangle,$$

where ${}_gM_{B,A_z} = {}_{g_2}M_{B,F_1} \diamond {}_{g_1}M_{F_1,A_z},\ 1 \le z \le k$. So, the data transformation of the resultant slice g is obtained by replacing the occurrence(s) of F_1 in $f_1(B_1,\ \ldots,\ B_p, F_1)$ by $f_2(A_1,\ \ldots,\ A_k)$. The dependence mapping between B and $A_z,\ 1 \le z \le k$, is obtained from ${}_{g_2}M_{B,F_1}$ and ${}_{g_1}M_{F_1,A_z}$ using transitive dependence computation over two sequences of statements. The transitive dependencies for the FIFOs from P_1 to P_2 can be obtained in a similar way. The following example illustrates the composition of ADDGs of KPN processes.

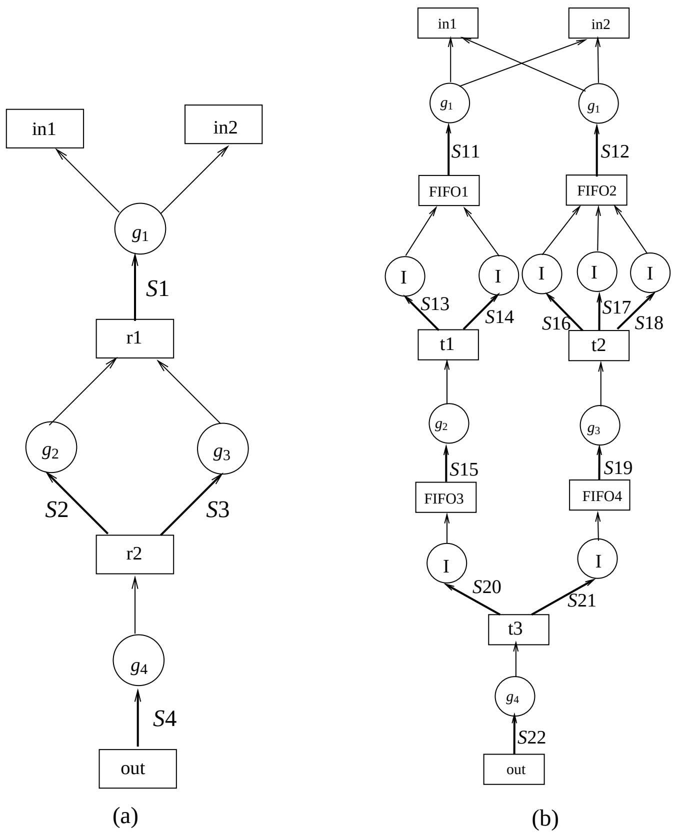


Figure 6.12: (a) The ADDG of the sequential behaviour given in example 21;
(b) The ADDG of the corresponding KPN behaviour

Example 28 Let us consider the ADDGs in figure 6.5. It may be recalled that these ADDGs are obtained from the KPN processes given in figure 6.4. Clearly, the ADDGs in figures 6.5(b) and 6.5(c) can be composed with the ADDG in figure 6.5(a). Similarly, the ADDG in 6.5(d) can be composed with the ADDGs in figures 6.5(b) and 6.5(c). The resultant ADDG of the KPN is depicted in figure 6.12(b). The ADDG of the original sequential behaviour (given in example 21) is depicted in figure 6.12(a). We have to establish the equivalence between these two ADDGs of figure 6.12.

Let us now compute the transitive dependence across the FIFO1 in the resultant ADDG. We have ${}_{S_{11}}M_{FIFO1,\ in1} = \{[f1In] \rightarrow [i, j] \mid f1In = (2\times i + j - 6) \wedge 1 \leq i \leq M \wedge 4 \leq j \leq N \wedge -i - j + 6 \geq 0 \wedge -i + M - 2 \geq 0\}$. We also have ${}_{S_{15}S_{13}}M_{FIFO3,\ FIFO1}$

$= \{[f3In] \rightarrow [f1Out] \mid f3In = (l+2\times m-3) \wedge f1Out = (l+2\times m-3) \wedge 3 \leq l \leq M \wedge 3 \leq n \leq N-1 \wedge l-4 \leq 0 \wedge m+l-7 \leq 0\}$. It may be noted that the value of $f1In$ in ${}_{S_{11}}M_{FIFO1,\ in1}$ and the values of $f3In$ and $f1Out$ in ${}_{S_{15}S_{13}}M_{FIFO3,\ FIFO1}$ are obtained by the method discussed above. Therefore, the transitive mapping from $FIFO3$ can be obtained by composing ${}_{S_{15}S_{13}}M_{FIFO3,\ FIFO1}$ with ${}_{S_{11}}M_{FIFO1,\ in1}$. Specifically, ${}_{S_{15}S_{13}S_{11}}M_{FIFO3,\ in1}$

$$\begin{aligned}
= \{&[f3In] \rightarrow [i,j] \mid \\
&1 \leq i \leq M \wedge 4 \leq j \leq N \wedge -i-j+6 \geq 0 \wedge -i+M-2 \geq 0 \wedge 3 \leq l \leq M \\
&\wedge 3 \leq n \leq N-1 \wedge l-4 \leq 0 \wedge m+l-7 \leq 0 \\
&\wedge \exists f1In, f1Out([f1In] \rightarrow [i,j] \in {}_{S_{11}}M_{FIFO1,\ in1} \\
&\wedge [f3In] \rightarrow [f1Out] \in {}_{S_{15}S_{13}}M_{FIFO3,\ FIFO1}) \\
&\wedge f3In = (l+2\times m-3) \wedge f1Out = (l+2\times m-3) \\
&\wedge f1In = (2\times i+j-6) \wedge f1In = f1Out\}.
\end{aligned}$$ □

6.3.4 Correctness of the composition operation

We now show that the composition operation correctly captures the data dependence of a deadlock free KPN. The (sequential to KPN) transformation process, however, may introduce deadlock in the generated KPN. In the next section, we shall show that deadlocks in KPN can be detected automatically during ADDG composition by carrying out some extra checks. Without loss of generality, we, therefore, assume here that the KPN is deadlock free to prove the correctness of the composition operation.

Theorem 20 *The ADDG composition mechanism captures correctly the data dependence across the deadlock free KPN processes.*

Proof: Let there be a FIFO F_1 from a KPN process P_1 to another KPN process P_2. Let G_1 and G_2 be the ADDGs of P_1 and P_2, respectively. Let S_i be a statement in P_1 defining F_1 and S_j be a statement in P_2 where F_1 is read into an array, A say.

Figure 6.11(a) depicts the IO-slice $g_1(F_1, A_1, \cdots, A_k)$ of G_1 ending in the write edge S_i. Similarly, figure 6.11(b) depicts the IO-slice $g_2(B, B_1, \cdots, F_1, \cdots, B_p)$ of G_2, where B is an output array of P_2; the statement S_j appears as a write edge corresponding to reading of the FIFO into the array A.

Let $g(C, I_1, \cdots, I_n)$ be a slice having the last write edge with statement labels S_t in an ADDG of a process P. Let π be any path from the start array C to some terminal array I_j, for some j, $1 \leq j \leq n$. Let the sequence of write edges in π be $\langle S_t, S_1, S_2, \ \ldots, \ S_k \rangle$, where S_t, S_1, $\ldots$, S_k are the statement labels. The following properties are satisfied by π.

Property 1: The sequence of statements $\langle S_t, S_1, S_2, \ \ldots, \ S_k \rangle$ in π gives the reverse execution sequence (not necessarily occurring consecutively in the process body) through which the output array C gets defined.

Property 2: Let $A_1, \ \ldots, \ A_k$ are the respective arrays, not necessarily distinct, defined by the statements $S_1, \ \ldots, \ S_k$. (The statement S_t defines the output array C.) All the elements of the arrays A_i, $1 \leq i \leq k$, and the output array C, assigned along the path π over the iteration domains of statements S_i and S_t get properly defined by virtue of having all the operand array elements defined before use in π. Thus, the dependence mapping ${}_{\pi}M_{C,I_j}$ is well defined.

If the composed ADDGs maintain these properties, then the composition process captures the dependence across the (deadlock free) KPN processes correctly.

In the composed ADDG of figure 6.11(c), let us, therefore, consider a path $\pi.S_i.\pi_1$ comprising the prefix π from the output array B to F_1 followed by the path $S_i.\pi_1$ in the slice g_1 from F_1 to some terminal array A_l of the slice g_1. (There may be more than one such path in the slice g_2 and hence in the composed ADDG G.)

Property 1 holds in the prefix π and the suffix $S_i.\pi_1$ of the path $\pi.S_i.\pi_1$ by construction of the ADDGs G_1 and G_2, respectively. Hence, the property holds for the entire path $\pi.S_i.\pi_1$ if it holds for the statement sequence $S_j.S_i$ in the path; this fact, however, is ensured by the FIFO operations *put* and *get* and by the *blocking read* operational semantics of KPN processes. It may be noted that the output array may not actually get defined if P_1 or P_2 or both cannot proceed because of deadlock.

Now, regarding property 2 for the path $S_i.\pi_1$, the dependence mappings ${}_{S_i.\pi_1}M_{F_1,A_1}, {}_{S_i.\pi_1}M_{F_1,A_2}, \cdots, {}_{S_i.\pi_1}M_{F_1,A_k}$, obtained during construction of the ADDG G_1, are well formed satisfying property 2. Hence, the mapping ${}_{S_i.\pi_1}M_{F_1,A_l}$ is well defined. So is the dependence mapping ${}_{\pi}M_{B,F_1}$ for the path π, ensured during construction of

```
for(i=0; i<50; i+=1)
   S1: FIFO1.put(f1(a[i]));
          process P1

for(i=0; i<60; i+=1)
   S2: b[i] = FIFO1.get();
for(i=0; i<50; i+=1)
   S3: FIFO2.put(f1(b[i]));
         process P2

for(i=0; i<50; i+=1)
   S4: c[i] = FIFO2.get();
       process P3
```

Figure 6.13: An example of producer-consumer deadlock

the ADDG G_2. Also, the composition step of G_1 and G_2 ensures that the mapping ${}_{\pi.S_i.\pi_1}M_{B,A_l}$ is well defined satisfying property 2 for the path $\pi.S_i.\pi_1$. Again, if P_1 or P_2 or both are in deadlock, the mapping ${}_{\pi.S_i.\pi_1}M_{B,A_l}$, although constructed by the composition process, will not actually be realized. □

Thus, the presence of deadlocks invalidate the composition of ADDGs. The following example illustrates the fact.

Example 29 Let us consider the KPN in figure 6.13. This KPN has a deadlock around the channel FIFO1 since the process P_1 puts 50 elements in FIFO1 whereas the process P_2 tries to get 60 elements from it. So, the process P_2 remains blocked on read from empty channel after consuming 50 elements from FIFO1. As a result, the second loop of P_2 and hence the process P_3 never execute. So, none of the elements of the array c gets defined.

Let us now discuss the composition mechanism in this KPN. It might be noted that each of the input and output pointers of the three FIFOs is the same as the corresponding loop index i in this example. The dependence mapping in P_1 is

$${}_{S1}M_{FIFO1,a} = \{[i] \rightarrow [i] \mid 0 \leq i \leq 49\}.$$

The dependence mappings in P_2 are

${}_{S2}M_{b,FIFO1} = \{[i] \rightarrow [i] \mid 0 \leq i \leq 59\}$. ${}_{S3}M_{FIFO2,b} = \{[i] \rightarrow [i] \mid 0 \leq i \leq 49\}$.

So, ${}_{S3S2}M_{FIFO2,FIFO1} = {}_{S3}M_{FIFO2,b} \diamond {}_{S2}M_{b,FIFO1} = \{[i] \rightarrow [i] \mid 0 \leq i \leq 49\}$.

The dependence mapping in P_3 is

${}_{S4}M_{c,FIFO2} = \{[i] \rightarrow [i] \mid 0 \leq i \leq 49\}$.

The composition of P_3 and P_2 around FIFO2 gives

${}_{S4S3S2}M_{c,FIFO1} = {}_{S4}M_{c,FIFO2} \diamond {}_{S3S2}M_{FIFO2,\ FIFO1} = \{[i] \rightarrow [i] \mid 0 \leq i \leq 49\}$.

The composition of composed behaviour of P_3 and P_2 with P_1 around FIFO1 gives

${}_{S4S3S2S1}M_{c,a} = {}_{S4S3S2}M_{c,FIFO1} \diamond {}_{S1}M_{FIFO1,a} = \{[i] \rightarrow [i] \mid 0 \leq i \leq 49\}$.

Therefore, the ADDG construction mechanism constructs the dependence mapping from the array *c* to *a* for the first 50 elements communicated through FIFO1 by ignoring the fact that second loop of P_2 and P_3 will never execute due to deadlock around FIFO1.

It may be noted that while computing (in the last step) the mapping ${}_{S4S3S2S1}M_{c,a}$, inferring deadlock from the fact that $|{}_{S1}U_a| < {}_{S4S3S2}M_{FIFO1}$ is not always correct. To illustrate this fact, let us assume that in figure 6.13, there is another loop body

```
for(i=0; i<9; i++)
  S5: FIFO1.put(f₂(d[i]));
```

after the first loop body in process P_1 in figure 6.13. In this case, the process P_1 puts 60 elements in FIFO1. Therefore, there will not be any deadlock in this KPN with this modification. However, the fact that $|{}_{S1}U_a| < {}_{S4S3S2}M_{FIFO1}$ is still true while computing the mapping ${}_{S4S3S2S1}M_{c,a}$. □

In general, therefore, the composition mechanism fails to to detect this kind of deadlocks. In the next section, we discuss how to detect deadlocks in KPNs.

6.4 Deadlock detection in a KPN

There may be two kinds of deadlocks in a KPN – global deadlocks and local deadlocks (Geilen and Basten, 2003). Recall that processes communicate through unbounded unidirectional FIFOs in a KPN model. The processes are synchronized by blocking read from empty FIFOs. A KPN is in a global deadlock if none of the processes in the network can make progress, that is, when all the processes are blocked reading from empty channels. In contrast, a *local deadlock* arises if some (but not all) of the processes in the network cannot progress and their further progress cannot be initiated by the rest of the network. Although KPN models use unbounded FIFOs, in practice, they are implemented using finite memory. Execution of a KPN behaviour can also stop if one or more processes are blocked writing to full channels. Therefore, some of the (local and global) deadlocks are *artificial deadlock* (Parks, 1995) in the sense that they do not exist in the actual KPN but arise due to finiteness of the FIFO implementation. A KPN with bounded FIFOs behaves in the same way as the corresponding KPN, except for the fact that certain write actions may be stalled by full channels. In general, the FIFO size cannot be decided statically (Buck, 1993). Therefore, the artificial deadlock situations are handled in runtime. The works reported in (Bharath et al., 2005; Cheung et al., 2009; Jiang et al., 2008; Li et al., 2010; Olson and Evans, 2005; Parks, 1995) speak of several such approaches for handling artificial deadlocks. The basic idea is to identify the artificial deadlock situations runtime. Once such a situation occurs, the size of the FIFO in question is increased by some fixed amount. As discussed in section 6.2, each process in the KPN is a nested loop program and the loop bounds are finite. For such a restricted class of KPNs, the amount of data communicated through a FIFO is finite and can be computed statically (Carpenter et al., 2010; Cheung et al., 2007). Therefore, artificial deadlock situation does not arise at runtime for these KPNs. A real deadlock, both local and global, however, can still be present in such KPNs. In the following, we show how the real deadlocks can be identified in such KPNs.

There are two factors leading to real deadlock in KPNs:

(i) *Insufficient communication of data elements over a FIFO:* The number of elements to be written in a FIFO by the producer process is less than the number of elements to be consumed from the FIFO.

(ii) *Circular dependencies among the processes:* If we consider the producer-consumer dependencies among the processes of a KPN, then there may be circular dependencies among the processes of a KPN. One process of such a cycle may be blocked if it tries to read from an empty channel. A process of this cycle must write some data to a FIFO in order for the cycle to make progress. This is impossible if at least one of the processes of the cycle can never progress. Therefore, circular dependencies among the processes may lead to real deadlock situation. This type of deadlock may be local if the cycle involves only some processes of the KPN and global if it involves all the processes.

One of the advantages of ADDG based modelling of the KPNs is that we can identify both kinds of real deadlock from their corresponding ADDGs. For the former case, we need to do some extra checking before compostion of the individual ADDGs of the processes. The seond one gets detected automatically during computation of transitive dependencies. In the following, we elaborate both of them.

6.4.1 Deadlock due to insufficient communication

Example 29 in the previous section is an instance of the class of deadlocks in question. The following extra processing is carried out for detection of such deadlocks. Let n_p be the number of elements to be written in a FIFO by the producer process and n_c be the number of elements to be consumed from the FIFO. If $n_p < n_c$ for any FIFO in a KPN, then the consumer process eventually gets blocked permanently and the execution of the KPN never terminates. Therefore, we need to ensure that $n_p \geq n_c$ for all FIFOs in a KPN. Let F be a FIFO from a producer process P to a consumer process C in a KPN. Let F be written by the statements i statements in P, S_i, $1 \leq i \leq k$. Therefore, the total number of data communicated by P to F, i.e., n_p, is the cardinality of $I_{S_1} \cup \ \ldots \ \cup I_{S_k}$. Similarly, let the elements of F be consumed by l statements, $T_j : \ 1 \leq j \leq l$. Therefore, the total number of data consumed from F by C, i.e., n_c, is the cardinality of $I_{T_1} \cup \ \ldots \ \cup I_{T_l}$. We obtain the values of n_p and n_c corresponding to each FIFO from ADDGs of the individual processes of a KPN and check whether $n_p \geq n_c$ or not. While $n_p < n_c$ helps detect the deadlock, if $n_p > n_c$, then some redundant data are communicated through the FIFO which will never be consumed by the consumer process; we, therefore, can report about the redundant communication over the FIFO. It may be noted that the values of n_p and n_c can only be computed if the lower limits and the upper limits and the step (increment) value of the loops of the processes are constant (i.e., they are not variable).

6.4.2 Deadlock due to circular dependence in a KPN

A cycle is formed in an ADDG when some statement, S say, defines an array such that definition of an element of the array depends on other elements of the same array defined by an earlier execution of the statement S. For example, S can be a statement of the form $S : a[i] = f(a[i-1])$. The producer-consumer relationship among the processes in a KPN may also result in a cycle in the corresponding ADDG. Presence of such ADDG cycles may result in deadlocks. We depict certain situations through the following examples before formalizing the detection mechanism for such deadlocks.

```
for(i=0; i<=10; i+=1) {
   S1: a[i] = FIFO2.get();
   S2: FIFO1.put(f1(a[i]);
   S3: c[i] = f3(a[i]); }
             process P1
```

```
for(i=0; i<=10; i+=1) {
   S4: b[i] = FIFO1.get();
   S5: FIFO2.put(f2(b[i]);
   S3: d[i] = f4(b[i]); }
             process P2
```

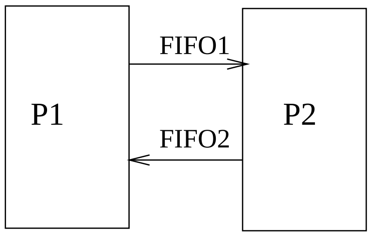


Figure 6.14: A KPN with circular dependence

Example 30 Let us consider the circular dependence of the KPN in figure 6.14. It may be noted from the behaviours of P1 and P2 that the i^{th} element communicated through FIFO1 depends on $a[i]$. This i^{th} element of FIFO1 is then used to define $b[i]$. The i^{th} element communicated through FIFO2 in turn depends on $b[i]$. Finally, $a[i]$ is defined by the i^{th} element communicated through FIFO2. There is a deadlock in this KPN because an ADDG cycle is formed by the statements $S1$, $S5$, $S4$ and $S2$ such that an element of $a[i]$ is defined in terms of some element of $b[i]$ which, in turn, depends on $a[i]$. In general, such circular dependence involves all the arrays defind in the cycle. □

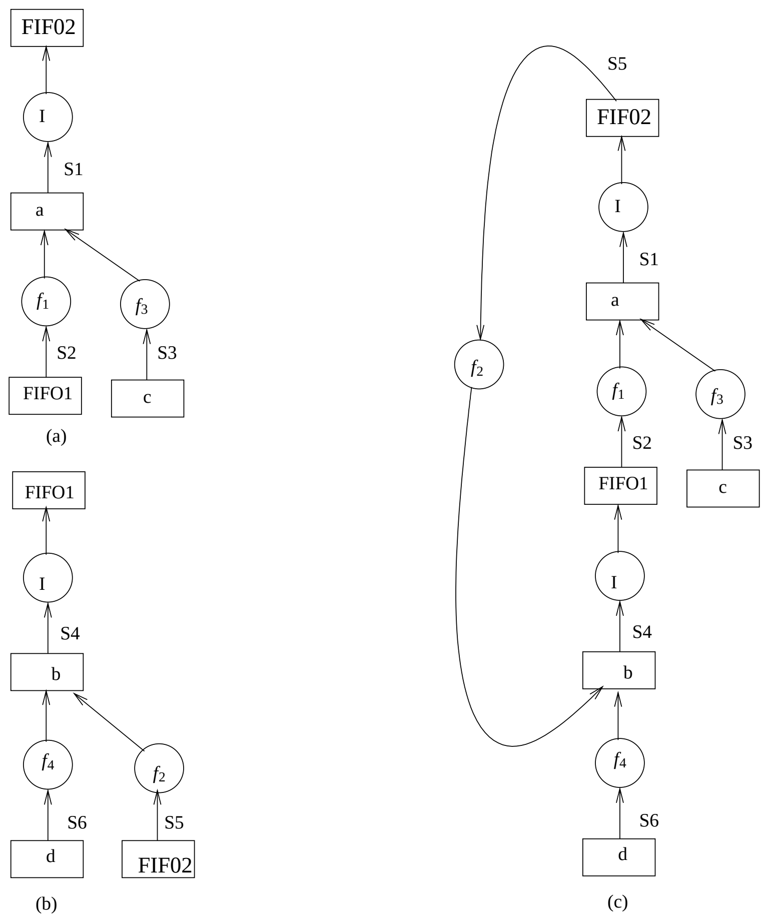
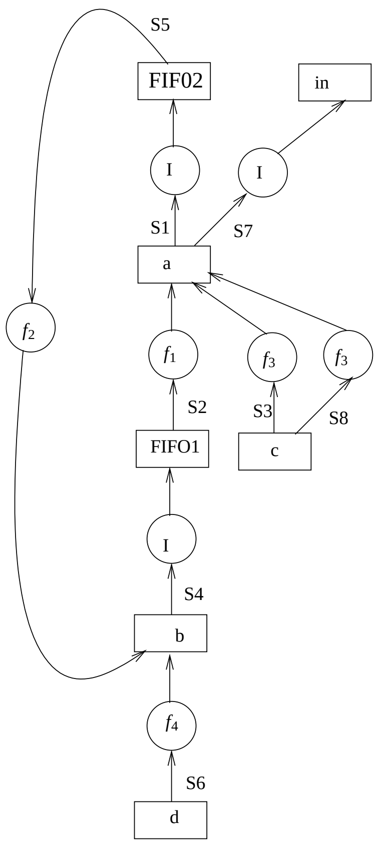


Figure 6.15: ADDGs of the processes in figure 6.14: (a) ADDG of process P1; (b) ADDG of process P2; (c) Composed ADDG of P1 and P2

The ADDGs of the processes $P1$ and $P2$ of the KPN of figure 6.14 are depicted in figures 6.15(a) and 6.15(b), respectively. The composed ADDG of P1 and P2 is shown figure 6.15(c). It may be noted from the ADDG in figure 6.15(c) that cycle has no outward edge. As discussed above, a cycle in an ADDG implies that an element of an array, a say, defined in one statement depends on the other elements of a defined by the earlier execution of the same statement. Therefore, the base element(s) of this array (i.e., the elements of a which do not depend on the other elements of a) must be defined by some other statement, S' say, in order to break this loop. Thus, S' creates an outward edge from the array node a in the ADDG. Therefore, a cycle in an ADDG without any outward edge is a sufficient condition for deadlock. However, presence of an ADDG cycle without any outward edge is not a necessary condition for deadlock. The following example provides an instance where deadlock occurs in spite of an outward edge from the ADDG cycle.

```
for(i=0; i<=10; i+=1) {
   S1: a[i] = FIFO2.get();
   S2: FIFO1.put($f_1$(a[i]));
   S3: c[i] = $f_3$(a[i]); }
for(i=11; i<=20; i+=1) {
   S4: a[i] = in[i];
   S5: c[i] = $f_3$(a[i]); }
             process P1

for(i=0; i<=10; i+=1) {
   S4: b[i] = FIFO1.get();
   S5: FIFO2.put($f_2$(b[i]));
   S3: d[i] = $f_4$(b[i]); }
             process P2
               (a)
```



(b)

Figure 6.16: Another KPN of the with circular dependence: (a) Processes of the KPN; (b) ADDG of the KPN

Example 31 Let us consider the KPN given in 6.16(a). This KPN is obtained from the KPN of figure 6.14 with following modification. In P1 in figure 6.16(a), we additionally define the elements of the array *a* from index 11 to 20 in terms of the elements of the input array *in*. We then use this elements to define the elements of the array *c* from index 11 to 20. The rest of the behaviours (of P1 and P2) is exactly the same as that of in figure 6.14. The ADDG of this KPN is depicted in figure 6.16(b). The first half of the behaviour of process P1 is not altered (where the deadlock actually occurs) by the latter part of P1. Clearly, the deadlock is still there in this modified KPN. □

It may be recalled from subsection 5.2.3 that a cycle, in which the elements of an array are defined in terms of earlier defined elements of the same array. In such a case, we compute the end-to-end mapping of the recurrence involved in the cycle. This end-to-end mapping is then used to compute the transitive dependencies across recurrence. Let us now compute the end-to-end mapping of the cycle in the ADDG in figure 6.15(c).

We have the following dependence mappings obtained from each statement of the KPN in figure 6.14 (and hence in figure 6.15(c)).

${}_{S1}M_{a,FIFO2} = \{[i] \rightarrow [i] \mid 0 \leq i \leq 10\}$,

${}_{S2}M_{FIFO1,a} = \{[i] \rightarrow [i] \mid 0 \leq i \leq 10\}$,

${}_{S3}M_{c,a} = \{[i] \rightarrow [i] \mid 0 \leq i \leq 10\}$,

${}_{S4}M_{b,FIFO1} = \{[i] \rightarrow [i] \mid 0 \leq i \leq 10\}$,

${}_{S5}M_{FIFO2,b} = \{[i] \rightarrow [i] \mid 0 \leq i \leq 10\}$ and

${}_{S6}M_{d,b} = \{[i] \rightarrow [i] \mid 0 \leq i \leq 10\}$.

It may be noted that FIFO pointers are represented as loop iterator i using the equation 6.5. The transitive dependence ${}_{S1S5S4S2}M_{a,a}$ in this cycle can be obtained by

$$\begin{aligned} {}_{S1S5S4S2}M_{a,a} &= {}_{S1}M_{a,FIFO2} \diamond {}_{S5}M_{FIFO2,b} \diamond {}_{S4}M_{b,FIFO1} \diamond {}_{S2}M_{FIFO1,a} \\ &= \{[i] \rightarrow [i] \mid 0 \leq i \leq 10\}. \end{aligned}$$

The transitive closure of this mapping is given by

$m = (\ {}_{S1S5S4S2}M_{a,a}\ )^{+} = \{[i] \rightarrow [P] \mid 0 \leq i \leq 10 \wedge P = \{\alpha \mid 0 \leq \alpha \leq i\}\}$.

The domain and range of m are

$d = \{[i] \mid 0 \leq i \leq 10\}$ and

$r = \{[\alpha] \mid 0 \leq \alpha \leq 10\}$.

The domain and range of end-to-end mapping are

$d' = (d - r) = \emptyset$, the empty set and

$r' = (r - d) = \emptyset$.

The domain and the range of the end-to-end mapping is empty in this case. As discussed in example 31, the statements involved in the cycle in the ADDG in figure

6.16(b) are identical to those involved in the cycle in figure 6.15(a). Therefore, the domain and the range of the end-to-end mapping of the cycle in figure 6.16(b) can also shown to be empty. Similarly, it can be shown that even for ADDG cycles without any outward edge (such as, the case depicted in figure 6.15) have empty domains and ranges for the end-to-end mappings.

The above discussion suggests that if there is a cycle in an ADDG corresponding to a KPN and the domain and range of the end-to-end mapping $M_{a,a}$ for any array a within the cycle is $\emptyset$, then we can ensure that there is a deadlock in the KPN. We have shown in subsection 6.3.3 that our method of composition of the ADDGs corresponding to the processes of a KPN also computes the transitive dependence mappings (and hence the end-to-end dependence mappings) accross processes during composition itself. Therefore, the deadlock due to circular dependencies among the processes in a KPN automatically gets detected during the composition steps.

6.5 Verification of KPN level transformations

In this section, we introduce several transformation techniques commonly used on a KPN behaviour. The objectives of this study include checking

1. how the ADDG of an initial KPN is changed by these transformations,
2. whether the current mechanism of converting a KPN to an ADDG works on the transformed KPN and
3. whether our equivalence checking method of ADDGs can verify all these transformations.

6.5.1 Channel merging

More than one channel from a producer process to a consumer process are merged into a single channel by channel merging transformations. All the data communicated between two processes through individual channels are now communicated through a single channel. Channel merging is relevant when a KPN behaviour is to be mapped on a multiprocessor system where a set of processors communicate through a shared bus and a shared memory. In such an architecture, communication and synchronization may be too costly to permit use of such redundant channels. Therefore, application of channel merging transformations reduce the control involved for communicating data over FIFOs and also reduce the total memory size allocated to the FIFOs in the case of communication of the same data elements over multiple FIFO channels.

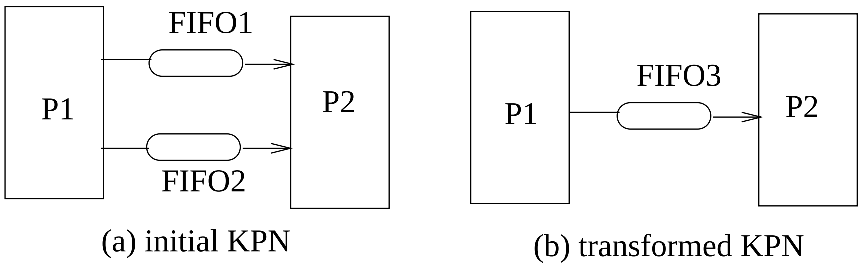


Figure 6.17: Channel Merging: (a) structure of the original KPN and (b) structure of transformed KPN

Let $FIFO1$ and $FIFO2$ be two FIFOs between the producer process P_1 and the consumer process P_2 in the initial KPN. The order in which the data are produced into $FIFO1$ and $FIFO2$ in the producer process may not be the same as the order in which data are consumed from them by the consumer process. Let us consider, for example, the KPNs in figure 6.17. Let the respective behaviours of the producer and the consumer process of the initial KPN and the transformed KPN of figure 6.17 be as shown in figure 6.18(a) and in figure 6.18(b), respectively. From figure 6.18(a), it may be noted that in the initial KPN, the producer process produces 100 elements of the $FIFO1$ first, the next 100 elements of $FIFO1$ and the first 100 elements of $FIFO2$ alternately next, and finally, the last 100 elements of $FIFO2$. The consumer process, on the other hand, consumes all 200 elements from $FIFO1$ first to compute the elements of the output array `out1` and then consumes all 200 elements from $FIFO2$ to compute the elements of the other output array `out2`. Clearly, the relative order of production of the elements of $FIFO1$ and $FIFO2$ in $P1$ is not the same as the relative consumption order from these two FIFOs in $P2$ in figure 6.18(a). In contrast, consider the situation, as depicted in figure 6.17(b), where the FIFOs $FIFO1$ and $FIFO2$ are merged into a single FIFO $FIFO3$ in the transformed KPN. Obviously, now that there is a single FIFO, the order of consumption of $FIFO3$ elements in P_2 should be the same as the production order into this FIFO. Thus, similar to process $P1$, the transformed consumer process P_2 now consumes first 100 elements from $FIFO3$ to compute the

```
for(i=0; i<100; i+=1)
   S1: FIFO1.put($f_1$(a[i+1]));
for(i=0; i<100; i+=1){
   S2: FIFO1.put($f_2$(a[i]));
   S3: FIFO2.put($f_3$(b[i+1]));}
for(i=0; i<100; i+=1)
   S4: FIFO2.put($f_4$(b[i]));
       process $P_1$

for(i=0; i<200; i+=1)
   T1: out1[i] = $f_5$(FIFO1.get());
for(i=0; i<200; i+=1)
   T2: out2[i] = $f_6$(FIFO2.get());
        Process $P_2$
             (a)
```

```
for(i=0; i<100; i+=1)
   S5: FIFO3.put($f_1$(a[i+1]));
for(i=0; i<100; i+=1){
   S6: FIFO3.put($f_2$(a[i]));
   S7: FIFO3.put($f_3$(b[i+1])); }
for(i=0; i<100; i+=1)
   S8: FIFO3.put($f_4$(b[i]));
    process $P_1$

for(i=0; i<100; i+=1)
   T3: out1[i] = $f_5$(FIFO3.get());
for(i=0; i<100; i+=1){
   T4: out1[i+100] = $f_5$(FIFO3.get());
   T5: out2[i] = $f_6$(FIFO3.get()); }
for(i=100; i<200; i+=1)
   T6: out2[i] = $f_6$(FIFO3.get());
    process $P_2$
            (b)
```

```
for(i=0; i<200; i+=1)
   T7: out1[i] = $f_5$(FIFO3.get());
for(i=0; i<200; i+=1)
   T8: out2[i] = $f_6$(FIFO3.get());
    Process $P_2{}'$
            (c)
```

Figure 6.18: Channel Merging: (a) behaviours of the processes in initial KPN with two processes and two FIFOs; (b) behaviours of the processes after merging two channels; (c) Erroneous behaviour of process P_2

first 100 elements of `out1`, then consumes next 200 elements from $FIFO3$ to compute the next 100 elements of `out1` and the first 100 elements `out2` alternately, and finally, consumes the last 100 elements to compute the next 100 elements of `out2`. If the process P_2 in the transformed KPN is retained in a form nearest to its behaviour in the initial KPN as shown in figure 6.18(c), then it may be noted that the arrays `out1` and `out2` are wrongly assigned. In the following, we explain how the composed ADDG for the KPN in figure 6.18(a) is found to be equivalent to the composed ADDG of the KPN in figure 6.18(b) and not equivalent to that of the KPN comprising $P1$ of figure 6.18(b) and P2 of figure 6.18(c).

In this example, in figure 6.18(a) $FIFO1$ elements are produced by the statements $S1$ and $S2$, $FIFO2$ elements by $S3$ and $S4$ and in figure 6.18(b) $FIFO3$ elements are

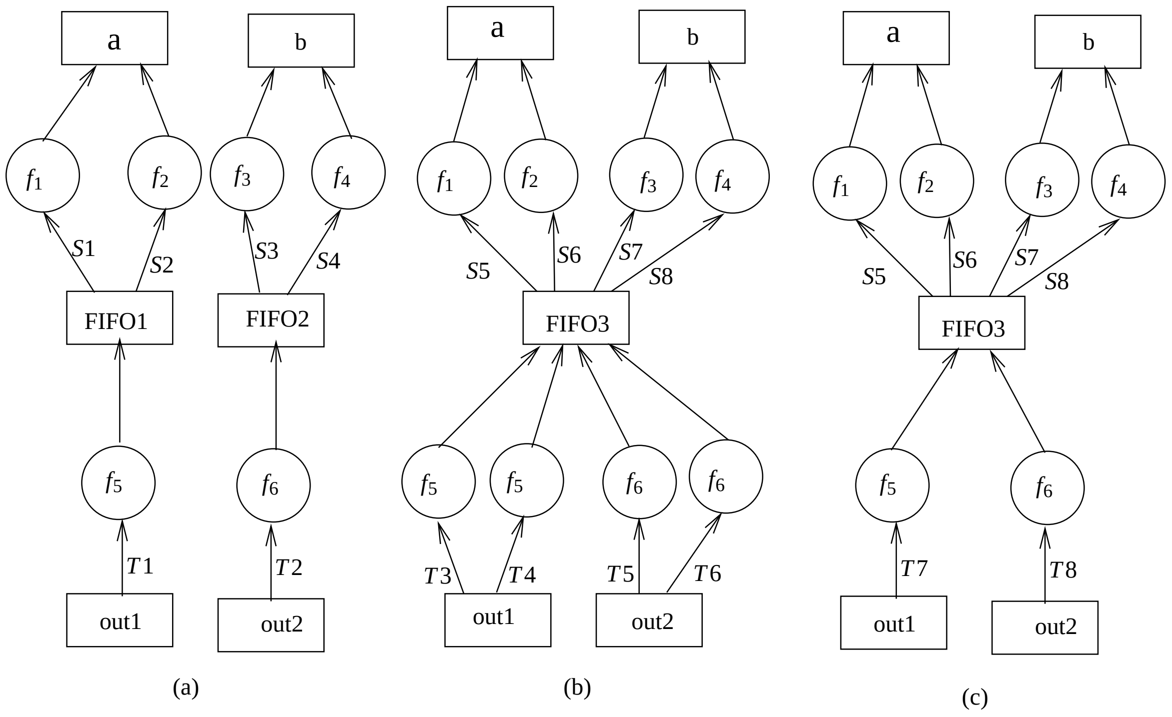


Figure 6.19: Channel Merging: (a) Composed ADDG of the initial KPN; (b) Composed ADDG of the transformed KPN; (c) Composed ADDG of the transformed KPN when P_2 is as in figure 6.18(c)

produced by $S5-S8$ and consumed by $T3-T6$. As discussed in subsection 6.3.2, we need to find the starting values of the corresponding FIFO pointers for each loop in these cases. These values can be obtained directly by the equation 6.5 since C_D is true. The dependence mappings are given next. We place the expressions (in terms of loop index i) of the in-pointers and the out-pointers in the dependence mappings.

The dependence mappings in the process P_1 of the initial KPN in figure 6.18(a) are as follows:

$_{S1}M_{FIFO1,a} = \{[i] \rightarrow [i+1] \mid 0 \leq i \leq 99\}$, where $FIFO1In = i$
$_{S2}M_{FIFO1,a} = \{[i+100] \rightarrow [i] \mid 0 \leq i \leq 99\}$, where $FIFO1In = i+100$
$_{S3}M_{FIFO2,b} = \{[i] \rightarrow [i+1] \mid 0 \leq i \leq 99\}$, where $FIFO2In = i$
$_{S4}M_{FIFO2,b} = \{[i+100] \rightarrow [i] \mid 0 \leq i \leq 99\}$, where $FIFO2In = i+100$.

The dependence mappings in the process P_2 of the initial KPN in figure 6.18(a) are as follows:

${}_{T1}M_{out1,FIFO1} = \{[i] \rightarrow [i] \mid 0 \leq i \leq 199\}$, where $FIFO1Out = i$
${}_{T2}M_{out2,FIFO2} = \{[i] \rightarrow [i] \mid 0 \leq i \leq 199\}$, where $FIFO2Out = i$.

The dependence mappings in the process P_1 of the transformed KPN in figure 6.18(b) are as follows:

${}_{S5}M_{FIFO3,a} = \{[i] \rightarrow [i+1] \mid 0 \leq i \leq 99\}$, where $FIFO3In = i$
${}_{S6}M_{FIFO3,a} = \{[2i+100] \rightarrow [i] \mid 0 \leq i \leq 99\}$, where $FIFO3In = 2i+100$
${}_{S7}M_{FIFO3,b} = \{[2i+101] \rightarrow [i+1] \mid 0 \leq i \leq 99\}$, where $FIFO3In = 2i+101$
${}_{S8}M_{FIFO3,b} = \{[i+300] \rightarrow [i] \mid 0 \leq i \leq 99\}$, where $FIFO3In = i+300$

It may be noted that the index expressions for the linear arrays corresponding to the FIFOs are obtained using the method describe in subsection 6.3.1. The details are omitted here for clarity in presentation. The dependence mappings in the process P_2 of the transformed KPN in figure 6.18(b) are as follows:

${}_{T3}M_{out1,FIFO3} = \{[i] \rightarrow [i] \mid 0 \leq i \leq 99\}$, where $FIFO3Out = i$
${}_{T4}M_{out1,FIFO3} = \{[i+100] \rightarrow [2i+100] \mid 0 \leq i \leq 99\}$, where $FIFO3Out = 2i+100$
${}_{T5}M_{out2,FIFO3} = \{[i] \rightarrow [2i+101] \mid 0 \leq i \leq 99\}$, where $FIFO3Out = 2i+101$
${}_{T6}M_{out2,FIFO3} = \{[i] \rightarrow [i+200] \mid 100 \leq i \leq 199\}$, where $FIFO3Out = i+200$.

The individual ADDGs of P_1 and P_2 of the KPN in figure 6.18(a) can be composed around $FIFO1$ and $FIFO2$. Similarly, the individual ADDGs of P_1 and P_2 of the KPN in figure 6.18(b) can be composed around $FIFO3$. The ADDGs of the initial and the transformed KPNs are shown in figure 6.19.

The transitive dependence mappings in the ADDG in figure 6.19(a) are as follows.

$$\begin{aligned}
{}_{T1S1}M_{out1,a} &= {}_{T1}M_{out1,FIFO1} \diamond {}_{S1}M_{FIFO1,a} \\
&= \{[i] \rightarrow [i] \mid 0 \leq i \leq 199\} \diamond \{[i] \rightarrow [i+1] \mid 0 \leq i \leq 99\} \\
&= \{[i] \rightarrow [i+1] \mid 0 \leq i \leq 99\} \\
{}_{T1S2}M_{out1,a} &= {}_{T1}M_{out1,FIFO1} \diamond {}_{S2}M_{FIFO1,a} \\
&= \{[i] \rightarrow [i] \mid 0 \leq i \leq 199\} \diamond \{[i+100] \rightarrow [i] \mid 0 \leq i \leq 99\} \\
&= \{[i] \rightarrow [i-100] \mid 100 \leq i \leq 199\}. \\
{}_{T2S3}M_{out2,b} &= {}_{T2}M_{out2,FIFO2} \diamond {}_{S3}M_{FIFO2,b} \\
&= \{[i] \rightarrow [i] \mid 0 \leq i \leq 199\} \diamond \{[i] \rightarrow [i+1] \mid 0 \leq i \leq 99\}. \\
&= \{[i] \rightarrow [i+1] \mid 0 \leq i \leq 99\}
\end{aligned}$$

$$\begin{aligned}
{}_{T2S4}M_{out2,b} &= {}_{T2}M_{out2,FIFO2} \diamond {}_{S4}M_{FIFO2,b} \\
&= \{[i] \to [i] \mid 0 \le i \le 199\} \diamond \{[i+100] \to [i] \mid 0 \le i \le 99\} \\
&= \{[i] \to [i-100] \mid 100 \le i \le 199\}.
\end{aligned}$$

The transitive dependence mappings in the ADDG in figure 6.19(b) are as follows.

$$\begin{aligned}
{}_{T3S5}M_{out1,a} &= {}_{T3}M_{out1,FIFO3} \diamond {}_{S5}M_{FIFO3,a} \\
&= \{[i] \to [i] \mid 0 \le i \le 99\} \diamond \{[i] \to [i+1] \mid 0 \le i \le 99\} \\
&= \{[i] \to [i+1] \mid 0 \le i \le 99\} \\
{}_{T4S6}M_{out1,a} &= {}_{T4}M_{out1,FIFO3} \diamond {}_{S6}M_{FIFO3,a} \\
&= \{[i+100] \to [2i+100] \mid 0 \le i \le 99\} \diamond \{[2i+100] \to [i] \mid 0 \le i \le 99\} \\
&= \{[i] \to [i-100] \mid 100 \le i \le 199\} \\
{}_{T5S7}M_{out2,b} &= {}_{T5}M_{out2,FIFO3} \diamond {}_{S7}M_{FIFO3,b} \\
&= \{[i] \to [2i+101] \mid 0 \le i \le 99\} \diamond \{[2i+101] \to [i+1] \mid 0 \le i \le 99\} \\
&= \{[i] \to [i+1] \mid 0 \le i \le 99\} \\
{}_{T6S8}M_{out2,b} &= {}_{T6}M_{out2,FIFO3} \diamond {}_{S8}M_{FIFO3,b} \\
&= \{[i] \to [i+200] \mid 100 \le i \le 199\} \diamond \{[i+300] \to [i] \mid 0 \le i \le 99\} \\
&= \{[i] \to [i-100] \mid 100 \le i \le 199\}
\end{aligned}$$

The operation $\diamond$ returns empty for other possible sequences of statements in the ADDG in figure 6.19(b). For example, let us compute the transitive dependence for the statement sequence $T3S7$.

$$\begin{aligned}
{}_{T3S7}M_{out1,b} &= {}_{T3}M_{out1,FIFO3} \diamond {}_{S7}M_{FIFO3,b} \\
&= \{[i] \to [i] \mid 0 \le i \le 99\} \diamond \{[2i+101] \to [i+1] \mid 0 \le i \le 99\} \\
&= \emptyset
\end{aligned}$$

Clearly, the operation $\diamond$ returns $\emptyset$ since the range of $FIFO3Out$ in ${}_{T3}M_{out1,FIFO3}$ and the domain of $FIFO3In$ in ${}_{S7}M_{FIFO3,b}$ are not overlapping.

It may be noted that the number of slices remains the same in both the ADDGs. In particular, there are four slices in each ADDG. Let us designate the slices of figure 6.19(a) as $g_{1i},\ 1 \le i \le 4$, and the same of figure 6.19(b) as $g_{2i},\ 1 \le i \le 4$. Each slice involves only one dependence mapping from an output array to an input array. The characteristic formulae of the slices $g_{1i},\ 1 \le i \le 4$, in the ADDG in figure 6.19(a) are $\tau_{11} = \langle f_5(f_1(a)), \langle\ {}_{T1S1}M_{out1,a}\rangle\rangle$,

$\tau_{12} = \langle f_5(f_2(a)), \langle\ _{T1S2}M_{out1,a}\rangle\rangle$,
$\tau_{13} = \langle f_6(f_3(b)), \langle\ _{T2S3}M_{out2,b}\rangle\rangle$ and
$\tau_{14} = \langle f_6(f_4(b)), \langle\ _{T2S4}M_{out2,b}\rangle\rangle$, respectively.

Similarly, the characteristic formulae of the slices $g_{2i},\ 1 \leq i \leq 4$, in the ADDG in figure 6.19(b) are
$\tau_{21} = \langle f_5(f_1(a)), \langle\ _{T3S5}M_{out1,a}\rangle\rangle$,
$\tau_{22} = \langle f_5(f_2(a)), \langle\ _{T4S6}M_{out1,a}\rangle\rangle$,
$\tau_{23} = \langle f_6(f_3(b)), \langle\ _{T5S7}M_{out2,b}\rangle\rangle$ and
$\tau_{24} = \langle f_6(f_4(b)), \langle\ _{T6S8}M_{out2,b}\rangle\rangle$, respectively.

The equivalence between the two the ADDGs can be shown by our method by showing that $g_{1i} \equiv g_{2i},\ 1 \leq i \leq 4$.

Let us now consider the erroneous behaviour of process P_2 of the transformed KPN in figure 6.18(c). The dependence mappings in this behaviour are as follows:

$$_{T7}M_{out1,FIFO3} = \{[i] \rightarrow [i] \mid 0 \leq i \leq 199\}$$
$$_{T8}M_{out2,FIFO3} = \{[i] \rightarrow [i+200] \mid 0 \leq i \leq 199\}$$

The composition of the ADDG of process P_2 in figure 6.18(c) with the ADDG of process P_1 in figure 6.18(b) results in the ADDG given in figure 6.19(c). We obtain the following transitive dependence mappings in the ADDG in figure 6.19(c):

$$\begin{aligned}
_{T7S5}M_{out1,a} = {} & _{T7}M_{out1,FIFO3} \diamond\ _{S5}M_{FIFO3,a} \\
= {} & \{[i] \rightarrow [i] \mid 0 \leq i \leq 199\} \diamond \{[i] \rightarrow [i+1] \mid 0 \leq i \leq 99\} \\
= {} & \{[i] \rightarrow [i+1] \mid 0 \leq i \leq 99\} \\
_{T7S6}M_{out1,a} = {} & _{T7}M_{out1,FIFO3} \diamond\ _{S6}M_{FIFO3,a} \\
= {} & \{[i] \rightarrow [i] \mid 0 \leq i \leq 199\} \diamond \{[2i+100] \rightarrow [i] \mid 0 \leq i \leq 99\} \\
= {} & \{[i] \rightarrow [i-100] \mid 100 \leq i \leq 199 \wedge \exists \alpha \in \mathbb{N}(i = 2\alpha)\} \\
_{T7S7}M_{out1,b} = {} & _{T7}M_{out1,FIFO3} \diamond\ _{S7}M_{FIFO3,b} \\
= {} & \{[i] \rightarrow [i] \mid 0 \leq i \leq 199\} \diamond \{[2i+101] \rightarrow [i+1] \mid 0 \leq i \leq 99\} \\
= {} & \{[i] \rightarrow [i-100] \mid 100 \leq i \leq 199 \wedge \exists \alpha \in \mathbb{N}(i = 2\alpha+1)\} \\
_{T8S6}M_{out2,a} = {} & _{T8}M_{out2,FIFO3} \diamond\ _{S6}M_{FIFO3,a} \\
= {} & \{[i] \rightarrow [i+200] \mid 0 \leq i \leq 199\} \diamond \{[2i+100] \rightarrow [i] \mid 0 \leq i \leq 99\} \\
= {} & \{[i] \rightarrow [i+200] \mid 0 \leq i \leq 99 \wedge \exists \alpha \in \mathbb{N}(i = 2\alpha)\} \\
_{T8S7}M_{out2,b} = {} & _{T8}M_{out2,FIFO3} \diamond\ _{S7}M_{FIFO3,b}
\end{aligned}$$

$$\begin{aligned}
&= \{[i] \rightarrow [i+200] \mid 0 \leq i \leq 199\} \diamond \{[2i+101] \rightarrow [i+1] \mid 0 \leq i \leq 99\} \\
&= \{[i] \rightarrow [i+200] \mid 0 \leq i \leq 99 \wedge \exists \alpha \in \mathbb{N}(i = 2\alpha+1)\} \\
{}_{T8S8}M_{out2,b} &= {}_{T8}M_{out2,FIFO3} \diamond {}_{S8}M_{FIFO3,b} \\
&= \{[i] \rightarrow [i+200] \mid 0 \leq i \leq 199\} \diamond \{[i+300] \rightarrow [i] \mid 0 \leq i \leq 99\} \\
&= \{[i] \rightarrow [i+200] \mid 100 \leq i \leq 199\}
\end{aligned}$$

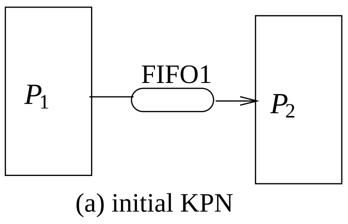
It may be noted that the operation $\diamond$ over the statement sequences $T7S8$ and $T8S5$ returns empty. There are six slices, g_{2i}, $1 \leq i \leq 6$, in this ADDG. The respective characteristic formulae of the slices are
$\tau_{21} = \langle f_5(f_1(a)), \langle\, {}_{T7S5}M_{out1,a}\rangle\rangle$,
$\tau_{22} = \langle f_5(f_2(a)), \langle\, {}_{T7S6}M_{out1,a}\rangle\rangle$,
$\tau_{23} = \langle f_5(f_3(b)), \langle\, {}_{T7S7}M_{out1,b}\rangle\rangle$,
$\tau_{24} = \langle f_6(f_2(a)), \langle\, {}_{T8S6}M_{out2,a}\rangle\rangle$,
$\tau_{25} = \langle f_6(f_3(b)), \langle\, {}_{T8S7}M_{out2,b}\rangle\rangle$ and
$\tau_{26} = \langle f_6(f_4(b)), \langle\, {}_{T8S8}M_{out2,b}\rangle\rangle$.

Our ADDG based equivalence checking between the ADDGs in figures 6.19(a) and 6.19(c) works as follows. The ADDG in figure 6.19(a) consists of four slices $\tau_{11}, \ldots, \tau_{14}$. Each slice in both the ADDGs form a slice class since each of the slices in an ADDG have a unique data transformation. Therefore, the ADDG in figure 6.19(a) consists of four slice classes whereas the ADDG in figure 6.19(c) consist of six slice classes. For the slice class τ_{11}, our algorithm finds τ_{21} as its equivalent slice class. It may be noted that the slice class τ_{22} of the ADDG figure 6.19(c) has the same data transformation (i.e., $f_5(f_2(a))$) as τ_{12} however the dependence mappings ${}_{T7S6}M_{out1,a}$ and ${}_{T1S6}M_{out1,a}$ are not the same. Therefore, for the slice class τ_{12}, the algorithm fails to find any equivalent slice class in the ADDG in figure 6.19(c). Hence, our algorithm reports that the ADDGs in figure 6.19(a) and in figure 6.19(c) are not equivalent.

We have shown here that our ADDG construction mechanism from KPN works when channel merging transformation is applied. Also, our ADDG based equivalence checking method is able to find the non-equivalence if the data dependence is violated.

6.5.2 Channel splitting

The channel splitting transformation breaks a channel between two processes in a KPN into more than one channel so that the data communicated through the original channel are now distributed over the split channels. The data parallelism over the processes are increased due to this transformation. Such transformation is relevant when parallel communication through multiple channels of parts of the data which are actually communicated serially through a single channel reduce the execution time of the KPN behaviour. Let us consider the KPNs in figure 6.20 where the channel $FIFO1$ of the initial KPN is split into the channels $FIFO2$ and $FIFO3$. The detailed behaviours of the KPNs are given in figure 6.21. In the initial KPN, all the 2000 elements are communicated sequentially through $FIFO1$, the first 1000 elements of which are stored in array $t1$ and the next 1000 elements are stored in array $t2$. The elements of arrays $t1$ and $t2$ are then used to define the elements of the output array out. In contrast, in the transformed KPN, the elements corresponding to the arrays $t1$ and $t2$ are communicated in parallel through channels $FIFO2$ and $FIFO3$. As a result, the process P_2 can compute the i^{th} element of the array out when the i^{th} elements of the channels $FIFO2$ and $FIFO3$ are available. Clearly, the execution time of the transformed KPN would be less than that of the initial KPN.



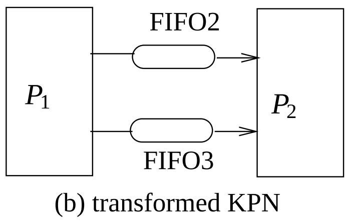


Figure 6.20: Channel Splitting: (a) initial KPN; (b) the transformed KPN

The ADDGs of the initial and the transformed KPN of figure 6.21 are shown in figure 6.22. In the following, we show that data input-output dependencies remain the same by checking equivalence of the ADDGs in figure 6.22.

The dependence mappings in the process P_1 of the initial KPN in figure 6.21(a) are as follows:

$_{S1}M_{FIFO1,a} = \{[i] \rightarrow [i] \mid 0 \leq i \leq 999\}$, where $FIFO1In = i$
$_{S2}M_{b,a} = \{[i] \rightarrow [i] \mid 0 \leq i \leq 999\}$

```
for(i=0; i<1000; i+=1)
   S1: FIFO1.put(f1(a[i]));
   S2: b[i] = f2(a[i]);
for(i=0; i<1000; i+=1)
   S3: FIFO1.put(f3(b[i]));
      process P1

for(i=0; i<1000; i+=1)
   S4: t1[i] = FIFO1.get();
for(i=0; i<1000; i+=1)
   S5: t2[i] = FIFO1.get();
for(i=0; i<1000; i+=1)
   S6: out[i] = f4(t1[i], t2[i]);
        process P2
             (a)
```

```
for(i=0; i<1000; i+=1)
   S7: FIFO2.put(f1(a[i]));
   S8: b[i] = f2(a[i]);
   S9: FIFO3.put(f3(b[i]));
      process P1

for(i=0; i<1000; i+=1)
   S10: out[i] = f4(FIFO2.get(),
                    FIFO3.get());
      process P2
         (b)
```

Figure 6.21: Channel Splitting: (a) A KPN with two processes and one channels; (b) The KPN after splitting the channel into two

${}_{S3}M_{FIFO1,b} = \{[i+1000] \rightarrow [i] \mid 0 \leq i \leq 999\}$, where $FIFO1In = i+1000$

The transitive dependence from FIFO1 to a in this process can be computed as follows:

$$ {}_{S3S2}M_{FIFO1,a} \;=\; {}_{S3}M_{FIFO1,b} \diamond {}_{S2}M_{b,a} \;=\; \{[i+1000] \rightarrow [i] \mid 0 \leq i \leq 999\} $$

The dependence mappings in the process P_2 of the initial KPN in figure 6.21(a) are as follows:

${}_{S4}M_{t1,FIFO1} = \{[i] \rightarrow [i] \mid 0 \leq i \leq 999\}$, where $FIFO1Out = i$
${}_{S5}M_{t2,FIFO1} = \{[i] \rightarrow [i+1000] \mid 0 \leq i \leq 999\}$, where $FIFO1Out = i+1000$
${}_{S6}M_{out,t1} = \{[i] \rightarrow [i] \mid 0 \leq i \leq 999\}$
${}_{S6}M_{out,t2} = \{[i] \rightarrow [i] \mid 0 \leq i \leq 999\}$

The transitive dependence mappings from *out* to $FIFO1$ in this process can be obtained as follows:

$$ {}_{S6S4}M_{out,FIFO1} \;=\; {}_{S6}M_{out,t1} \diamond {}_{S4}M_{t1,FIFO1} \;=\; \{[i] \rightarrow [i] \mid 0 \leq i \leq 999\} $$
$$ {}_{S6S5}M_{out,FIFO1} \;=\; {}_{S6}M_{out,t2} \diamond {}_{S5}M_{t2,FIFO1} \;=\; \{[i] \rightarrow [i+1000] \mid 0 \leq i \leq 999\} $$

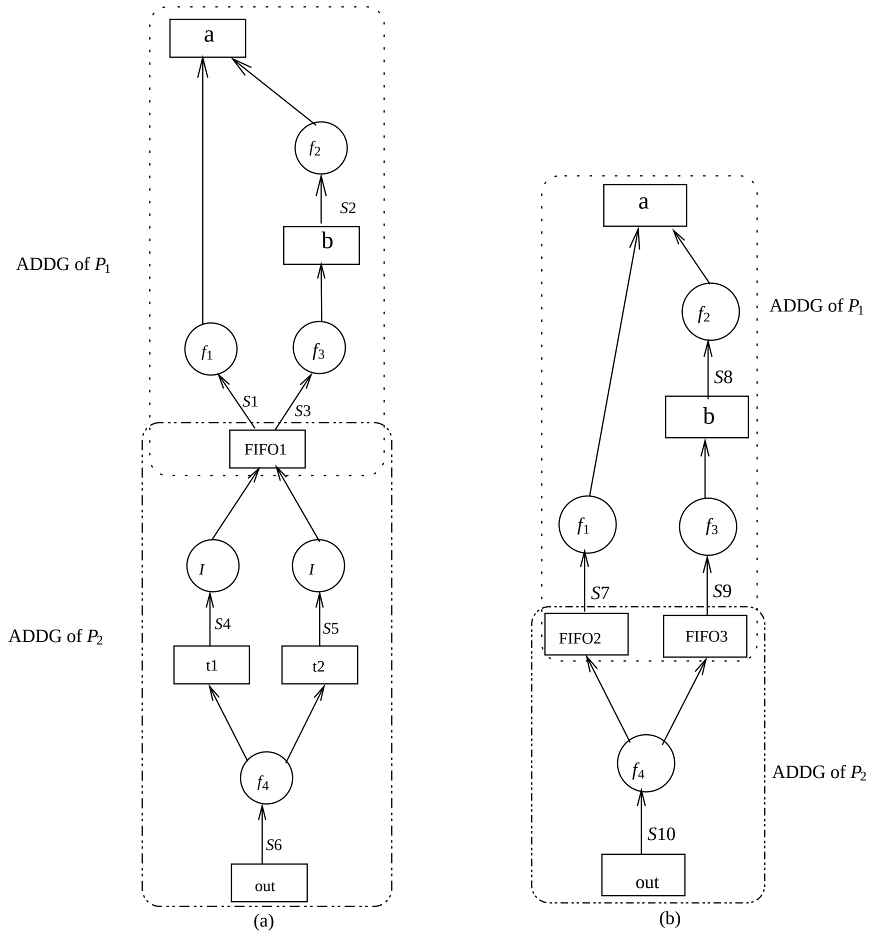


Figure 6.22: Channel Splitting: (a) Composed ADDG of the input KPN; (b) Composed ADDG of the transformed KPN

For figure 6.21(b), the dependence mappings in the process P_1 of the transformed KPN are as follows:

$_{S7}M_{FIFO2,a} = \{[i] \rightarrow [i] \mid 0 \leq i \leq 999\}$, where $FIFO2In = i$
$_{S8}M_{b,a} = \{[i] \rightarrow [i] \mid 0 \leq i \leq 999\}$
$_{S9}M_{FIFO3,b} = \{[i] \rightarrow [i] \mid 0 \leq i \leq 999\}$, where $FIFO3In = i$

The transitive dependence from $FIFO3$ to a can be obtained as follows:

$_{S9S8}M_{FIFO3,a} = \{[i] \rightarrow [i] \mid 0 \leq i \leq 999\}$

In figure 6.21(b), the dependence mappings in the process P_2 of the transformed KPN are as follows:

${}_{S10}M_{out,FIFO2} = \{[i] \rightarrow [i] \mid 0 \leq i \leq 999\}$, where $FIFO2Out = i$
${}_{S10}M_{out,FIFO3} = \{[i] \rightarrow [i] \mid 0 \leq i \leq 999\}$, where $FIFO3Out = i$

The individual ADDGs of P_1 and P_2 of the KPN in figure 6.21(a) can be composed around $FIFO1$. Similarly, the individual ADDGs of P_1 and P_2 of the KPN in figure 6.21(b) can be composed around $FIFO2$ and $FIFO3$. The ADDGs of the initial and the transformed KPNs are shown in figure 6.22.

The output array *out* to the input array *a* dependence mappings in the ADDG in figure 6.22(a) are as follows:

$$\begin{aligned}
{}_{S6S4S1}M_{out,a} &= {}_{S6S4}M_{out,FIFO1} \diamond {}_{S1}M_{FIFO1,a} \\
&= \{[i] \rightarrow [i] \mid 0 \leq i \leq 999\} \diamond \{[i] \rightarrow [i] \mid 0 \leq i \leq 999\} \\
&= \{[i] \rightarrow [i] \mid 0 \leq i \leq 999\} \\
{}_{S6S5S3S2}M_{out,a} &= {}_{S6S5}M_{out,FIFO1} \diamond {}_{S3S2}M_{FIFO1,a} \\
&= \{[i] \rightarrow [i+1000] \mid 0 \leq i \leq 999\} \diamond \{[i+1000] \rightarrow [i] \mid 0 \leq i \leq 999\} \\
&= \{[i] \rightarrow [i] \mid 0 \leq i \leq 999\}
\end{aligned}$$

It may be noted that the operation $\diamond$ returns empty for statement sequences $S6S5S1$ and $S6S4S3S2$. Following transitive dependence computations elaborate this fact:

$$\begin{aligned}
{}_{S6S4S3S2}M_{out,a} &= {}_{S6S4}M_{out,FIFO1} \diamond {}_{S3S2}M_{FIFO1,a} \\
&= \{[i] \rightarrow [i] \mid 0 \leq i \leq 999\} \diamond \{[i+1000] \rightarrow [i] \mid 0 \leq i \leq 999\} \\
&= \emptyset
\end{aligned}$$

$$\begin{aligned}
{}_{S6S5S1}M_{out,a} &= {}_{S6S5}M_{out,FIFO1} \diamond {}_{S1}M_{FIFO1,a} \\
&= \{[i] \rightarrow [i+1000] \mid 0 \leq i \leq 999\} \diamond \{[i] \rightarrow [i] \mid 0 \leq i \leq 999\} \\
&= \emptyset
\end{aligned}$$

The output array *out* to the input array *a* dependence mappings in the ADDG in figure 6.22(b) are as follows.

$$\begin{aligned}
{}_{S10S7}M_{out,a} &= {}_{S10}M_{out,FIFO2} \diamond {}_{S7}M_{FIFO2,a} \\
&= \{[i] \rightarrow [i] \mid 0 \leq i \leq 999\} \diamond \{[i] \rightarrow [i] \mid 0 \leq i \leq 999\} \\
&= \{[i] \rightarrow [i] \mid 0 \leq i \leq 999\} \\
{}_{S10S9S8}M_{out,a} &= {}_{S10}M_{out,FIFO3} \diamond {}_{S9S8}M_{FIFO3,b} \\
&= \{[i] \rightarrow [i] \mid 0 \leq i \leq 999\} \diamond \{[i] \rightarrow [i] \mid 0 \leq i \leq 999\}
\end{aligned}$$

$$= \{[i] \rightarrow [i] \mid 0 \leq i \leq 999\}.$$

There is one slice in each of the ADDGs. The characteristic formula of the slice in figure 6.22(a) is $\tau_1 = \langle f_4(f_1(a), f_3(f_2(a))), \langle\ _{S6S4S1}M_{out,a},\ _{S6S5S3S2}M_{out,a}\rangle\rangle$. It may be noted that both the mappings are from the array *out* to the array *a*. Therefore, we need to put them in a proper order so that equivalence between the mappings with corresponding slice can be established. The simplification rules for normalized dependence mappings presented in subsection 5.4.2 helps us to achieve that. The characteristic formula of the slice in figure 6.22(b) is $\tau_2 = \langle f_4(f_1(a), f_3(f_2(a))), \langle\ _{S10S7}M_{out,a},\ _{S10S9S8}M_{out,a}\rangle\rangle$. Our ADDG based equivalence checking method shows that $\tau_1 \equiv \tau_2$.

The channel merging transformation is usually used in combination with process splitting transformation which will be discussed in the next subsection.

6.5.3 Process splitting

An arbitrary process of a KPN behaviour can be split into a number of autonomously running processes by applying process splitting transformation. Process splitting transformation increases the parallelism in the KPN (Turjan, 2007). It also helps distribute the workload of a single process over multiple processes to better balance the network which results in improvement in the total execution time of the KPN (Meijer et al., 2009). Let us consider the KPNs, for example, in figure 6.23 where the KPN in figure 6.23(b) is obtained from the KPN in figure 6.23(a) by splitting its process P_2 into processes P_{2a} and P_{2b}. The behaviours of the processes are given in figure 6.24. It may be noted that the process P_2 of the initial KPN is computation intensive compared to the process P_1 since P_1 has 100 iterations of the loop to execute whereas P_2 has 5000 iterations to execute. The process P_2 is split into processes P_{2a} and P_{2b} such that the even iterations of the outer loop iterator (i.e., i) are now executed in process P_{2a} and the odd iterations of the outer loop iterator (i.e., i) are now executed in process P_{2b}. The corresponding transformed KPN has now two processes in place of P_2 computing the output *out* resulting in a more balanced network. More importantly, processes P_{2a} and P_{2b} are executed in parallel in the transformed KPN. In most of the cases, splitting a process causes splitting of its associated channels. The FIFO $FIFO1$ in the KPN in figure 6.23(a), for example, is also split into $FIFO2$ and $FIFO3$ along with process P_2 in the transformed KPN.

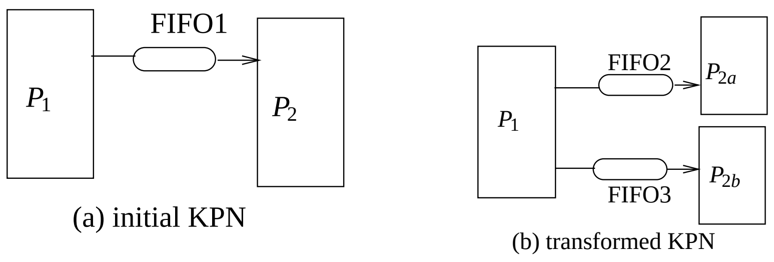


Figure 6.23: Process Splitting: (a) initial KPN; (b) transformed KPN after process splitting

The process splitting transformation has the following two effects in a KPN behaviour: (i) *Iteration domain of a statement is partitioned into sub-domains and the sub-domains in the partition are placed in the processes obtained after splitting*, and (ii) *Two data dependent statements are placed in two different processes obtained after splitting*. In the following, we elaborate these two cases, highlight the verification goals for them and then show that those verification goals can be achieved using our ADDG based equivalence checking method.

Case (i): In course of process splitting transformations, the iteration domains of some statements are partitioned into sub-domains which are placed in the processes obtained after splitting. In this case, the statement in question occurs (in duplication) in each of the processes obtained after splitting. Since, the iteration domain of the original statement is partitioned over many processes, we need to ensure that the iteration domain of the original statement is equal to the union of the iteration domains of its instances in the split processes. The example in figure 6.24 reflects this situation. Here, the iteration domains of the statements $S2$, $S3$ and $S4$ of the process P_2 are partitioned into two sub-domains and are placed in the processes P_{2a} and P_{2b} such that the even iterations of the outer loop iterator (i.e., i) of the process P_2 are executed in the process P_{2a} and the odd iterations of the same are executed in the process P_{2b}. Specifically, the statements $S7$ and $S10$ of the transformed consumer processes P_{2a} and P_{2b}, respectively, are the two instances of $S2$ of the original consumer process P_2. Similarly, $S8$ and $S11$ (of P_{2a} and P_{2b}, respectively) are the two instances of the statement $S3$ of P_2 and $S9$ and $S12$ (of P_{2a} and P_{2b}, respectively) are the two instances of $S4$ in P_2. Let us now discuss how the equivalence of these two KPNs are established by our method.

```
for(i=0; i<100; i+=1)
  S1: FIFO1.put(f1(a[i]));
      process P1

for(i=0; i<100; i+=1){
  S2: t[i] = FIFO1.get();
  for(j=0; j<50; j+=1){
    if(i < 25)
      S3: out[i][j] = f2(t[i]);
    else
      S4: out[i][j] = f3(t[i]);
}}
    process P2
             (a)
```

```
for(i=0; i<100; i+=1){
  if(i%2==0)
    S5: FIFO2.put(f1(a[i]));
  else
    S6: FIFO3.put(f1(a[i])); }
     process P1

for(i=0; i<100; i+=2){
  S7: t1[i] = FIFO2.get();
  for(j=0; j<50; j+=1){
    if(i < 25)
      S8: out[i][j] = f2(t1[i]);
    else
      S9: out[i][j] = f3(t1[i]);
}}
     process P2a

for(i=1; i<100; i+=2){
  S10: t2[i] = FIFO3.get();
  for(j=0; j<50; j+=1){
    if(i < 25)
      S11: out[i][j] = f2(t2[i]);
    else
      S12: out[i][j] = f3(t2[i]);
}}
     process P2b
          (b)
```

Figure 6.24: Process Splitting: (a) A KPN with two processes and one channel; (b) The KPN after splitting the process P_2 into two processes P_{2a} and P_{2b}

The dependence mapping in the process P_1 of the initial KPN in figure 6.24(a) is as follows:

$$_{S1}M_{FIFO1,a} = \{[i] \rightarrow [i] \mid 0 \le i \le 99\}.$$

In figure 6.24(a), the dependence mappings in the process P_2 of the initial KPN are as follows:

$$_{S3S2}M_{out,FIFO1} = \{[i][j] \rightarrow [i] \mid 0 \le i \le 24 \wedge 0 \le j \le 49\}$$
$$_{S4S2}M_{out,FIFO1} = \{[i][j] \rightarrow [i] \mid 25 \le i \le 99 \wedge 0 \le j \le 49\}$$

In figure 6.24(b), the dependence mappings in the process P_1 of the transformed KPN are as follows:

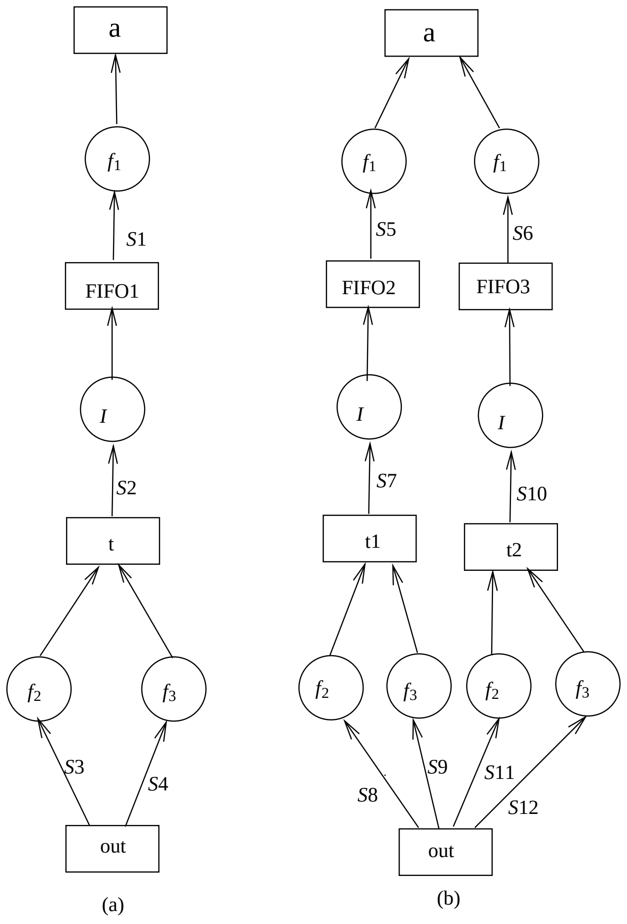


Figure 6.25: Process Splitting: (a) ADDG of the input KPN and (b) ADDG of the transformed KPN

$_{S5}M_{FIFO2,a} = \{[i] \rightarrow [i] \mid 0 \leq i \leq 99 \wedge i\%2 == 0\}$
$_{S6}M_{FIFO3,a} = \{[i] \rightarrow [i] \mid 0 \leq i \leq 99 \wedge i\%2 \neq 0\}$

In figure 6.24(b), the dependence mappings in the process P_{2a} of the transformed KPN are as follows:

$_{S8S7}M_{out,FIFO2} = \{[i][j] \rightarrow [i] \mid 0 \leq i \leq 24 \wedge 0 \leq j \leq 49 \wedge \exists \alpha \in \mathbb{N}(i = 2\alpha)\}$
$_{S9S7}M_{out,FIFO2} = \{[i][j] \rightarrow [i] \mid 25 \leq i \leq 99 \wedge 0 \leq j \leq 49 \wedge \exists \alpha \in \mathbb{N}(i = 2\alpha)\}$

In figure 6.24(b), the dependence mappings in the process P_{2b} of the transformed KPN are as follows:

$_{S11S10}M_{out,FIFO3} = \{[i][j] \rightarrow [i] \mid 1 \leq i \leq 24 \wedge 0 \leq j \leq 49 \wedge \exists \alpha \in \mathbb{N}(i = 2\alpha + 1)\}$
$_{S12S10}M_{out,FIFO3} = \{[i][j] \rightarrow [i] \mid 25 \leq i \leq 99 \wedge 0 \leq j \leq 49 \wedge \exists \alpha \in \mathbb{N}(i = 2\alpha + 1)\}$

The individual ADDGs of P_1 and P_2 of the KPN in figure 6.24(a) can be composed around $FIFO1$. The individual ADDGs of P_1 and P_{2a} of the KPN in figure 6.24(b) can be composed around $FIFO2$. Similarly, the ADDGs of P_1 and P_{2b} of the KPN in figure 6.24(b) can be composed around $FIFO3$. The ADDGs of the initial and the transformed KPNs are shown in figure 6.25.

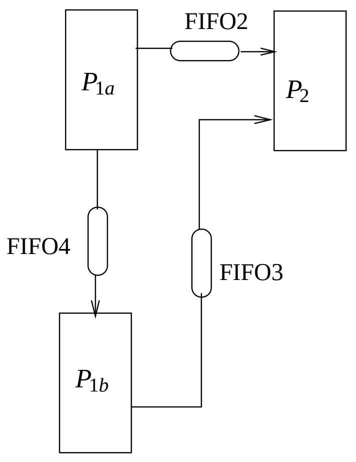
The output to input dependence mappings in the ADDG in figure 6.25(a) are as follows.

$$\begin{aligned}
{}_{S3S2S1}M_{out,a} &= {}_{S3S2}M_{out,FIFO1} \diamond {}_{S1}M_{FIFO1,a} \\
&= \{[i][j] \rightarrow [i] \mid 0 \leq i \leq 24 \wedge 0 \leq j \leq 49\} \\
{}_{S4S2S1}M_{out,a} &= {}_{S4S2}M_{out,FIFO1} \diamond {}_{S1}M_{FIFO1,a} \\
&= \{[i][j] \rightarrow [i] \mid 25 \leq i \leq 99 \wedge 0 \leq j \leq 49\}
\end{aligned}$$

The output to input dependence mappings in the ADDG in figure 6.25(b) are as follows.

$$\begin{aligned}
{}_{S8S7S5}M_{out,a} &= {}_{S8S7}M_{out,FIFO2} \diamond {}_{S5}M_{FIFO2,a} \\
&= \{[i][j] \rightarrow [i] \mid 0 \leq i \leq 24 \wedge 0 \leq j \leq 49 \wedge \exists \alpha \in \mathbb{N}(i = 2\alpha)\} \\
{}_{S9S7S5}M_{out,a} &= {}_{S9S7}M_{out,FIFO2} \diamond {}_{S5}M_{FIFO2,a} \\
&= \{[i][j] \rightarrow [i] \mid 25 \leq i \leq 99 \wedge 0 \leq j \leq 49 \wedge \exists \alpha \in \mathbb{N}(i = 2\alpha)\} \\
{}_{S11S10S6}M_{out,a} &= {}_{S11S10}M_{out,FIFO3} \diamond {}_{S6}M_{FIFO3,a} \\
&= \{[i][j] \rightarrow [i] \mid 1 \leq i \leq 24 \wedge 0 \leq j \leq 49 \wedge \exists \alpha \in \mathbb{N}(i = 2\alpha + 1)\} \\
{}_{S12S10S6}M_{out,a} &= {}_{S9S7}M_{out,FIFO2} \diamond {}_{S5}M_{FIFO2,a} \\
&= \{[i][j] \rightarrow [i] \mid 25 \leq i \leq 99 \wedge 0 \leq j \leq 49 \wedge \exists \alpha \in \mathbb{N}(i = 2\alpha + 1)\}.
\end{aligned}$$

There are two slices, g_{11} and g_{12} in the ADDG in figure 6.25(a). Their respective characteristic formula of the slices are $\tau_{11} = \langle f_2(f_1(a)), \langle {}_{S3S2S1}M_{out,a} \rangle \rangle$ and $\tau_{12} = \langle f_3(f_1(a)), \langle {}_{S4S2S1}M_{out,a} \rangle \rangle$. There are four slices g_{2i}, $1 \leq i \leq 4$, in the ADDG in figure 6.25(b). Their respective characteristic formula of the slices are $\tau_{21} = \langle f_2(f_1(a)), \langle {}_{S8S7S5}M_{out,a} \rangle \rangle$, $\tau_{22} = \langle f_3(f_1(a)), \langle {}_{S9S7S5}M_{out,a} \rangle \rangle$, $\tau_{23} = \langle f_2(f_1(a)), \langle {}_{S11S10S6}M_{out,a} \rangle \rangle$ and $\tau_{24} = \langle f_3(f_1(a)), \langle {}_{S12S10S6}M_{out,a} \rangle \rangle$. The data transformation of slices τ_{21} and τ_{23} are the same. Hence they form a slice class, say g_{21}. The characteristic formula of g_{21} is $\tau_{g_{21}} = \langle f_2(f_1(a)), \langle {}_{g_{21}}M_{out,a} \rangle \rangle$, where ${}_{g_{21}}M_{out,a} = {}_{S8S7S5}M_{out,a} \cup {}_{S11S10S6}M_{out,a}$. So, our algorithm establishes that $\tau_{g_{21}} \equiv \tau_{11}$. Similarly, the slices τ_{22} and τ_{24} form a slice class, g_{22} say, which is shown to be the equivalent of τ_{12} by our method. Therefore, the two ADDGs in figure 6.25 are equivalent.



(a) KPN after process splitting

```
for(i=0; i<1000; i+=1){
   S7′: FIFO2.put(f1(a[i]));
   S8′: FIFO4.put(f2(a[i]));}
      process P1a

for(i=0; i<1000; i+=1)
   S9′: FIFO3.put(f2(FIFO4.get()));
      process P1b

for(i=0; i<1000; i+=1)
  S10′: out[i] = f3(FIFO2.get(), FIFO3.get());
     process P2
```

(b) behaviours of the processes

Figure 6.26: The KPN after splitting process P_1 of KPN in figure 6.21; (b) into two process P_{1a} and P_{1b}

Case (ii): Two data dependent statements (of an original process), S and S', say, the latter having dependence on the former, are placed in two different processes obtained after splitting. In this case, therefore, a FIFO is to be added from the process that contains S to the process containing S' after splitting.

Let us consider the KPN in figure 6.21(b). The process P_1 of this KPN is split into two process P_{1a} and P_{1b}. The structure of the KPN after splitting is shown in figure 6.26(a) and the behaviours of the processes are shown in figure 6.26(b). Here, the statements corresponding to $S7$ and $S8$ of the process P_1 of KPN in figure 6.21(b) are placed in the process P_{1a} of the transformed KPN in figure 6.26(b) and the statement corresponding to $S9$ of the process P_1 is placed in the process P_{1b}. It may be noted that the statement $S9$ is dependent on the statement $S8$ since $S8$ computes the elements of the array b which are used in $S9$. Therefore, a channel $FIFO4$ is added between processes P_{1a} and P_{1b} to communicate these elements.

The ADDG of the KPN in figure 6.26 would be the same as the ADDG (in figure 6.22(b)) of its initial KPN with the array node b replaced by $FIFO4$. So, the number of slices of the original ADDGs remains the same in the transformed ADDG in this case. Only the computation of transitive dependence from *out* to a for the statement sequence $S10'S9'S8'$ would be different from the ADDG in figure 6.22(b). Similarly, when two data independent statements are placed in two different processes obtained after splitting, no FIFO is required between those processes. So, the ADDGs of the initial and the transformed KPNs would be identical. Therefore, the number of slices of the ADDG obtained by composition of the ADDGs of the processes after splitting would be the same as the number of slices of the ADDG of the original KPN process in both the cases. Our equivalence checking method is able to show equivalence in such case.

6.5.4 Process merging

The process merging transformation reduces the number of processes in a KPN by serializing some processes in a single compound process. This transformation may be applied when (i) the number of processes is larger than the number of processing elements, or ii) the network is not well balanced and therefore the same overall performance can be achieved using less resources (Meijer et al., 2010b). To illustrate the second case, let three be parallel processes $P1, P2$ and $P3$ say, in a KPN. Let $P1$ be a computation intensive process and takes much higher time to complete its execution compared to $P2$ and $P3$. In this case, even if the parallelism decreases in the KPN due to merging of two less computation intensive processes $P2$ and $p3$, the overall execution time may not differ due to the computation intensive process $P1$. However, FIFO communication cost among less computation intensive processes is eliminated by merging them. A process merging example is given in figure 6.27. Here, the processes P_2 and P_3 of the initial KPN are merged into a single process. The detailed behaviours of the processes are shown in figure 6.28.

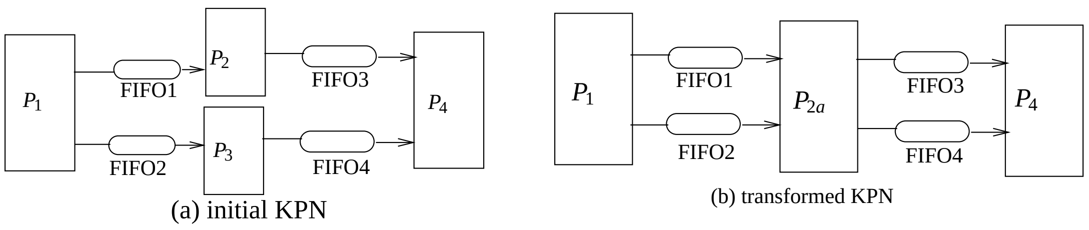


Figure 6.27: Process Merging: (a) the initial KPN; (b) the transformed KPN

The composed ADDGs of the initial and the transformed KPNs in figure 6.28 are shown in figures 6.29(a) and 6.29(b), respectively. Even though the number of processes decreases by process merging transformation, the number of FIFO channels and the number of statements remain the same in both the KPNs. Therefore, the ADDGs of both the KPNs have exactly identical structure in this case.

```
for(i=0; i<100; i+=1){
  if(i%2 == 0)
      S1: FIFO1.put(f1(a[i]));
  S2: FIFO2.put(f2(a[i])); }
      process P1

for(i=0; i<50; i+=1)
  S3: FIFO3.put(f3(FIFO1.get()));
      process P2

for(i=0; i<100; i+=1)
  S4: FIFO4.put(f4(FIFO2.get()));
      process P3

for(i=0; i<100; i+=2){
  S5: out[i] = f5(FIFO3.get());
  S6: out[i+1] = f5(FIFO4.get()); }
for(i=100; i<150; i+=1)
  S7: out[i] = f5(FIFO4.get());
      process P4
                (a)
```

```
for(i=0; i<100; i+=1){
  if(i%2 == 0)
     S3′: FIFO3.put(f3(FIFO1.get()));
   S4′: FIFO4.put(f4(FIFO2.get()));}
      process P2a
                (b)
```

Figure 6.28: Process Merging: A KPN with four processes and four channels; (b) The KPN comprising P_1, P_{2a} and P_4 obtained after merging two processes P_2 and P_3 (into P_{2a})

The individual loop bodies of the merged process may again be combined into a single loop body. This may reduce the overall execution time of the transformed KPN since the number of iterations of the loops of the merged process is decreased by loop merging. An erroneous loop merging, however, may alter the cardinalities of the iteration domains of some of the statements which, in turn, results in (i) deadlock in the KPN or (ii) some of the output elements remain undefined. For both cases, therefore, it is required to ensure that the iteration domains of the statements of the transformed process are not altered by process merging.

In figure 6.28, for example, the loop bodies of process P_2 and P_3 are merged into a single loop body in the merged process P_{2a}. In case the `if` statement is missed out in the process P_{2a} by mistake, the cardinality of the iteration domain of the statement $S3'$ (i.e., $|I_{S3'}|$) would contain 100 elements in processes P_{2a} whereas $|I_{S3}|$ for $S3$ in the original process P_2 contains 50 elements causing a deadlock in the transformed KPN. The deadlock in the KPN can be identified by our method given in section 6.4.

Let us consider another erroneous situation where the statement $S4'$ is placed

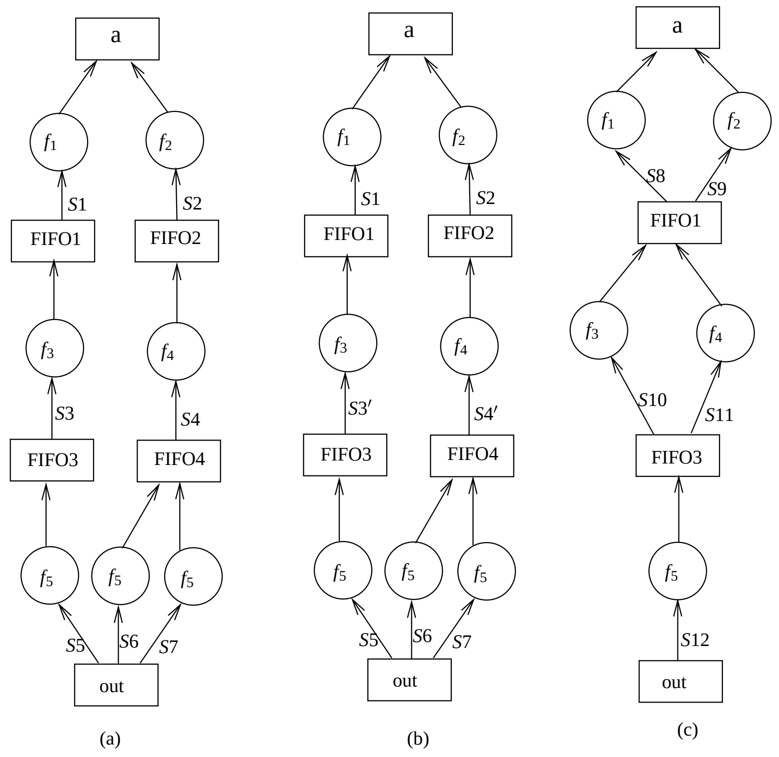


Figure 6.29: Process Merging: (a) ADDG of the input KPN; (b) ADDG of the transformed KPN; (c) ADDG of the KPN in figure 6.30

within the scope of `if` statement in the process P_{2a}. In this case, $|I_{S4'}| = 50$ whereas $|I_{S4}| = 100$. If an output array is defined in the statement $S4'$ (instead of communicating data to $FIFO4$), then 50 elements of that output array remain undefined in the transformed KPN. This kind of errors can be identified by our ADDG based equivalence checking method as follows: the domains of the transitive dependencies from that output array to the input arrays includes only those elements which are defined. Consequently, our method finds a slice g_1 starting from that output array in the ADDG of the original KPN for which there would exist a slice g_2 in the ADDG of the transformed KPN such that the domains of dependence mappings in g_2 are the subset of the domains of the dependence mappings in g_1. Therefore, our method can show the non-equivalence for the incorrect cases of process merging transformations.

Merging processes sometimes create scope of applying channel merging. The channels $FIFO1$ and $FIFO2$ and also the channels $FIFO3$ and $FIFO4$ in figure 6.28(b), for example, can be merged. The transformed KPN after merging channels $FIFO1$ and $FIFO2$ into $FIFO2$ and channels $FIFO3$ and $FIFO4$ into $FIFO3$ of the

```
for(i=0; i<100; i+=1){
   if(i%2 == 0){
        S8: FIFO1.put(f1(a[i]));}
   S9: FIFO1.put(f2(a[i])); }
       process P1

for(i=0; i<100; i+=1){
   if(i%2 == 0){
        S10: FIFO3.put(f3(FIFO1.get()));}
   S11: FIFO3.put(f4(FIFO1.get()));}
       process P2

for(i=0; i<150; i+=1)
   S12: out[i] = f5(FIFO3.get());
       process P4
```

Figure 6.30: KPN after merging channels FIFO1 and FIFO2 into FIFO2 and channels FIFO3 and FIFO4 into FIFO3

KPN of figure 6.28(b) is shown in figure 6.30. The ADDG of this KPN is shown in figure 6.29(c). Working of our method for channel merging transformations has been discussed in subsection 6.5.1. It can be shown that our method can establish the equivalence for this case also. The details of the working of our method for this example are omitted for brevity of presentation.

6.5.5 Message vectorization

This transformation vectorizes (or buffers) the messages communicated between two processes, thereby reducing the number of inter process communications (Fei et al., 2007). The number of messages reduces significantly by this transformation. This transformation is effective as long as the energy overheads of buffering do not outweigh the gains of reducing the number of messages communicated. Let us consider, for example, the KPNs in figure 6.31. There are 100 messages (each of size 1) sent from P_1 to P_2 through channel $FIFO1$. In the transformed KPN of figure 6.31(b), obtained by applying message vectorization to this KPN, in the process P_1, the messages are first vectorized or buffered with vector size 10. They are then transmitted through $FIFO1$ to process P_2 in a burst. Let us assume that $putline(M,k)$ and $getline(k)$ be the version of *put* and *get*, respectively, that captures the communication of messages of size k. Specifically, $F.putline(M,k)$ indicates that k elements starting from location

M are put into the FIFO F. Similarly, $M = F.getline(k)$ indicates that k elements are obtained from the FIFO F into the location starting from M. The number of messages in the transformed KPN is, therefore, 10 (each of size 10). The number of messages are reduced by this transformation. Therefore, it is required to ensure that the total volume of messages passed between the two processes remains the same as in the original KPN.

```
for(i=0; i<10; i+=1)
 for(j=0; j<10; j+=1)
   S1: FIFO1.put($f_1$(a[i][j]));
     process $P_1$

for(i=0; i<10; i+=1)
 for(j=0; j<10; j+=1)
   S2: out[i][j]=$f_2$(FIFO1.get());
     process $P_2$
             (a)
```

```
for(i=0; i<10; i+=1){
  for(j=0; j<10; j++)
    S3: line[i][j] = $f_1$(a[i][j]);
  S4: FIFO1.putline(line[i], 10);
                 //whole line }
       process $P_1$

for(i=0; i<10; i+=1){
  S5: tmp[i] = FIFO1.getline(10);
               //get a row into tmp
  for(j=0; j<10; j++)
    S6: out[i][j] = $f_2$(tmp[i][j]);}
       process $P_2$
          (b)
```

Figure 6.31: Message Vectorization: (a) initial KPN with two processes and one channel; (b) The KPN when message is sent as a vector of 10 elements

```
for(i=0, f1In=-1; i<10; i+=1){
   for(j=0; j<10; j++)
      S3: line[i][j] = $f_1$(a[i][j]);
   for(k=0; k<10; k++) //in place of S4: FIFO1.putline(line[i], 10);
      S7: FIFO1[++f1In] = line[i][k]; }
       process $P_1$

for(i=0, f1Out=-1; i<10; i+=1){
   for(k=0; k<10; k++)  //in place of S5: tmp[i] = FIFO1.getline(10);
       S8: tmp[i][k] = FIFO1[++f1Out];
   for(j=0; j<10; j++)
       S6: out[i][j] = $f_2$(tmp[i][j]);}
       process $P_2$
```

Figure 6.32: Message de-vectorization: Modified behaviour of the KPN in figure 6.31(b)

For the *put* and *get* primitives the message size is one. Therefore, the in-pointer and the out-pointer of the linear array corresponding to a FIFO are increased by one

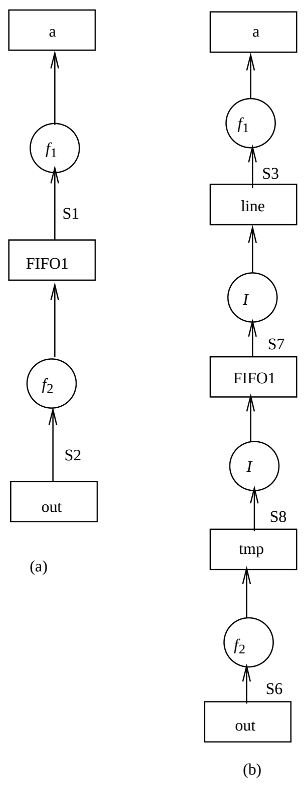


Figure 6.33: Message de-vectorization: (a) ADDG of the input KPN; (b) ADDG of the transformed KPN

for these primitives, respectively. The current ADDG construction mechanism can be enhanced for the *getline* and *putline* primitives as follows to handle message vectorization. The *putline* (*getline*) primitive is de-vectorized by replacing it by a loop body with k iterations each involving put (get) primitive of one element to (from) the linear array. The modified behaviours of the processes of the KPN in figure 6.31(b), for example, is given in figure 6.32. With this modification, our method can handle message vectorization. In the following, we show the details of the working of our method for the example given in 6.31.

The dependence mapping in the process P_1 of the initial KPN in figure 6.31(a) is as follows:

$$_{S1}M_{FIFO1,a} = \{[10*i+j] \rightarrow [i][j] \mid 0 \leq i \leq 9 \wedge 0 \leq j \leq 9\}.$$

The dependence mapping in the process P_2 of the initial KPN in figure 6.31(a) is as follows:

$${}_{S2}M_{out,FIFO1} = \{[i][j] \rightarrow [10*i+j] \mid 0 \leq i \leq 9 \, \wedge \, 0 \leq j \leq 9\}.$$

Since the relation between $FIFO1In$ ($FIFO1Out$) and the loop indices (i, j) are linear and there is additional condition here, the relation can be obtained by the equation 6.5. The ADDG given in figure 6.33(a) of the KPN in figure 6.31(a) can be obtained by composing the ADDGs of P_1 and P_2 around $FIFO1$. Therefore, the dependence between *out* and *a* in the ADDG in figure 6.33(a) is

$$\begin{aligned} {}_{S2S1}M_{out,a} &= {}_{S2}M_{out,FIFO1} \diamond {}_{S1}M_{FIFO1,a} \\ &= \{[i][j] \rightarrow [i][j] \mid 0 \leq i \leq 9 \, \wedge \, 0 \leq j \leq 9\}. \end{aligned}$$

The dependence mappings in the process P_1 of the transformed KPN in figure 6.32 which is semantically equivalent to the transformed KPN of figure 6.31(b) are as follows:

$$\begin{aligned} &{}_{S3}M_{line,a} = \{[i][j] \rightarrow [i][j] \mid 0 \leq i \leq 9 \wedge 0 \leq j \leq 9\} \\ &{}_{S7}M_{FIFO1,line} = \{[10*i+k] \rightarrow [i][k] \mid 0 \leq i \leq 9 \wedge 0 \leq k \leq 9\} \end{aligned}$$

$FIFO1In$ is $10*i+k$ in ${}_{S7}M_{FIFO1,line}$. Again, this relation can be obtained by the equation 6.5.

The transitive dependence between $FIFO1$ and a can be obtained by

$$\begin{aligned} {}_{S7S3}M_{FIFO1,a} &= {}_{S7}M_{FIFO1,line} \diamond {}_{S3}M_{line,a} \\ &= \{[10*i+j] \rightarrow [i][j] \mid 0 \leq i \leq 9 \wedge 0 \leq j \leq 9\}. \end{aligned}$$

The dependence mappings in the process P_2 of the transformed KPN in figure 6.32 are as follows:

$$\begin{aligned} &{}_{S8}M_{tmp,FIFO1} = \{[i][k] \rightarrow [10*i+k] \mid 0 \leq i \leq 9 \wedge 0 \leq k \leq 9\} \\ &{}_{S6}M_{out,tmp} \;\; = \{[i][j] \rightarrow [i][j] \mid 0 \leq i \leq 9 \wedge 0 \leq j \leq 9\} \end{aligned}$$

The transitive dependence between *out* and $FIFO1$ can be obtained by

$$\begin{aligned} {}_{S6S8}M_{out,FIFO1} &= {}_{S6}M_{out,tmp} \diamond {}_{S8}M_{tmp,FIFO1} \\ &= \{[i][j] \rightarrow [10*i+j] \mid 0 \leq i \leq 9 \wedge 0 \leq j \leq 9\} \end{aligned}$$

The ADDG of the KPN in figure 6.32 is depicted in figure 6.33(b) which is obtained by composing the ADDGs of P_1 and P_2 around $FIFO1$. Therefore, the dependence between *out* and *a* in the ADDG in figure 6.33(b) is

$$\begin{aligned} {}_{S6S8S7S3}M_{out,a} &= {}_{S6S8}M_{out,FIFO1} \diamond {}_{S7S3}M_{FIFO1,a} \\ &= \{[i][j] \rightarrow [i][j] \mid 0 \leq i \leq 9 \wedge 0 \leq j \leq 9\}. \end{aligned}$$

Each of the ADDGs consists of one slice. The characteristic formula of the slice, g_1 say, of the ADDG in figure 6.33(a) is $\tau_{g_1} = \langle f_2(f_1(a)), \langle {}_{S2S1}M_{out,a} \rangle \rangle$. The characteristic formula of the slice, g_2 say, of the ADDG in figure 6.33(b) is $\tau_{g_2} = \langle f_2(f_1(a)), \langle {}_{S6S8S7S3}M_{out,a} \rangle \rangle$. Omega calculator used in our method can established the equivalence of ${}_{S6S8S7S3}M_{out,a}$ and ${}_{S2S1}M_{out,a}$. Hence, our method shows that $g_1 \equiv g_2$.

6.5.6 Computation migration

This transformation relocates computations from one process to another so that the overheads of synchronization and interprocess communication get reduced (Fei et al., 2007). The number of processes remains the same under this transformation. However, the number of communication channels between processes may change. Some of the channels of the original process are removed and some new channels are added to the KPN. The primary motivation of this transformation is to reduce the number of channels or the total volume of communication over all the channels.

Let us consider the KPNs in figure 6.34. The initial KPN consists of three processes and four FIFO channels. Each rectangular box within a process defines an array. In figure 6.34(a), the FIFO $FIFO1$ from the array a of the process $P1$ to the array c of the process $P2$ indicates that the elements of a is defined in process $P1$, then is communicated through $FIFO1$ to the process $P2$ and then is used to define the elements of the array c. All such edges between processes in the KPNs in figure 6.34 have similar meaning. In figure 6.34(b) where both a and c reside in the process $P1$, the directed edge from a to c simply indicates that the elements of the array a are used to define the elements of the array c. All such edges with the processes in the KPNs in figure 6.34 are interpreted likewise. In short, the directed edges capture the define-use relation between the arrays. The transformed behaviour after computation migration is shown in figure 6.34(b). It may be noted that the computation defining the array c is migrated from process $P2$ to process $P1$. As a result, the FIFOs $FIFO1$ and $FIFO3$ are not required and hence removed. A new FIFO $FIFO5$, however, is to be added

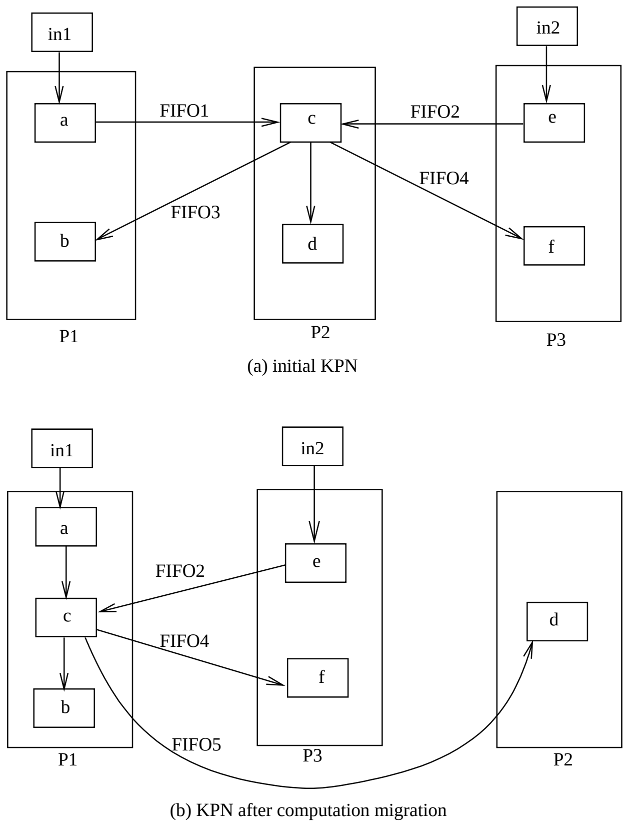


Figure 6.34: Computation Migration: (a) the structure of the initial KPN; (b) the structure of the KPN after computation migrations

between process $P1$ and $P3$. The detailed behaviours of the processes of these two KPNs are given in figure 6.35.

The ADDG of the initial and the transformed KPNs of figure 6.35 obtained by our method are depicted in figure 6.36. Both the ADDGs in figure 6.36 have three slices. They are $g_{11}(b, \langle in1, in2\rangle)$, $g_{12}(d, \langle in1, in2\rangle)$ and $g_{13}(f, \langle in1, in2\rangle)$ in the ADDG in figure 6.36(a) and $g_{21}(b, \langle in1, in2\rangle)$, $g_{22}(d, \langle in1, in2\rangle)$ and $g_{23}(f, \langle in1, in2\rangle)$ in the ADDG in figure 6.36(b). Since some FIFOs are removed and some new FIFOs are introduced in the transformed KPN by computation migration transformation, the computation of the transitive dependencies from the output arrays to the inputs arrays would be different in the ADDGs. Our ADDG based method establishes the equivalence between the ADDGs as follows.

The characteristic formulas of the slices in ADDG in figure 6.36(a) are

$\tau_{g_{11}} = \langle f_2(f_3(f_1(in1), f_5(in2))), \langle\ {}_{S3S6S4S2S1}M_{b,in1},\ {}_{S3S6S4S9S8}M_{b,in2}\rangle\rangle,$

```
for(i=0; i<100; i+=1){
   S1: a[i] = f1(in1[i]);
   S2: FIFO1.put(a[i]);
   S3: b[i] = f2(FIFO3.get()); }
       process P1

for(i=0; i<100; i+=1){
   S4: c[i] = f3(FIFO1.get(),
                 FIFO2.get());
   S5: d[i] = f4(c[i]);
   S6: FIFO3.put(c[i]);
   S7: FIFO4.put(c[i]); }
       process P2

for(i=0; i<100; i+=1){
   S8: e[i] = f5(in2[i]);
   S9: FIFO2.put(e[i]);
   S10: f[i] = f6(FIFO4.get()); }
       process P3

             (a)
```

```
for(i=0; i<100; i+=1){
  S1: a[i] = f1(in1[i]);
  S4: c[i] = f3(a[i], FIFO2.get());
  S3: b[i] = f2(c[i]);
  S11: FIFO4.put(c[i]);
  S12: FIFO5.put(c[i]); }
      process P1

for(i=0; i<100; i+=1){
  S5: d[i] = f4(FIFO5.get()); }
      process P2

for(i=0; i<100; i+=1){
  S8: e[i] = f5(in2[i]);
  S9: FIFO2.put(e[i]);
  S10: f[i] = f6(FIFO4.get()); }
      process P3

             (b)
```

Figure 6.35: Computation Migration: (a) A KPN with three processes and four channels; (b) The KPN after migration of computation f_3 from process P2 to P1

$\tau_{g_{12}} = \langle f_4(f_3(f_1(in1), f_5(in2))), \langle\, _{S5S4S2S1}M_{b,in1},\ _{S5S4S9S8}M_{b,in2} \rangle\rangle$ and
$\tau_{g_{13}} = \langle f_6(f_3(f_1(in1), f_5(in2))), \langle\, _{S10S7S4S2S1}M_{b,in1},\ _{S10S7S4S9S8}M_{b,in2} \rangle\rangle$, where all the mappings are of the form $\{[i]-> [i] \mid 0 \leq i \leq 99\}$. The details of computation of the dependence mappings of the slices are intuitive and hence, are not shown explicitly.

The characteristic formulas of the slices in ADDG in figure 6.36(b) are

$\tau_{g_{21}} = \langle f_2(f_3(f_1(in1), f_5(in2))), \langle\, _{S3S4S1}M_{b,in1},\ _{S3S4S9S8}M_{b,in2} \rangle\rangle$,
$\tau_{g_{22}} = \langle f_4(f_3(f_1(in1), f_5(in2))), \langle\, _{S5S12S4S1}M_{b,in1},\ _{S5S12S4S9S8}M_{b,in2} \rangle\rangle$ and
$\tau_{g_{23}} = \langle f_6(f_3(f_1(in1), f_5(in2))), \langle\, _{S10S11S4S1}M_{b,in1},\ _{S10S11S4S9S8}M_{b,in2} \rangle\rangle$, where all the mappings are of the form $\{[i]-> [i] \mid 0 \leq i \leq 99\}$. The details of computation of the dependence mappings are not shown explicitly for the same reason. Clearly, $g_{1k} \equiv g_{2k},\ 1 \leq k \leq 4$. Hence, the two ADDGs are equivalent.

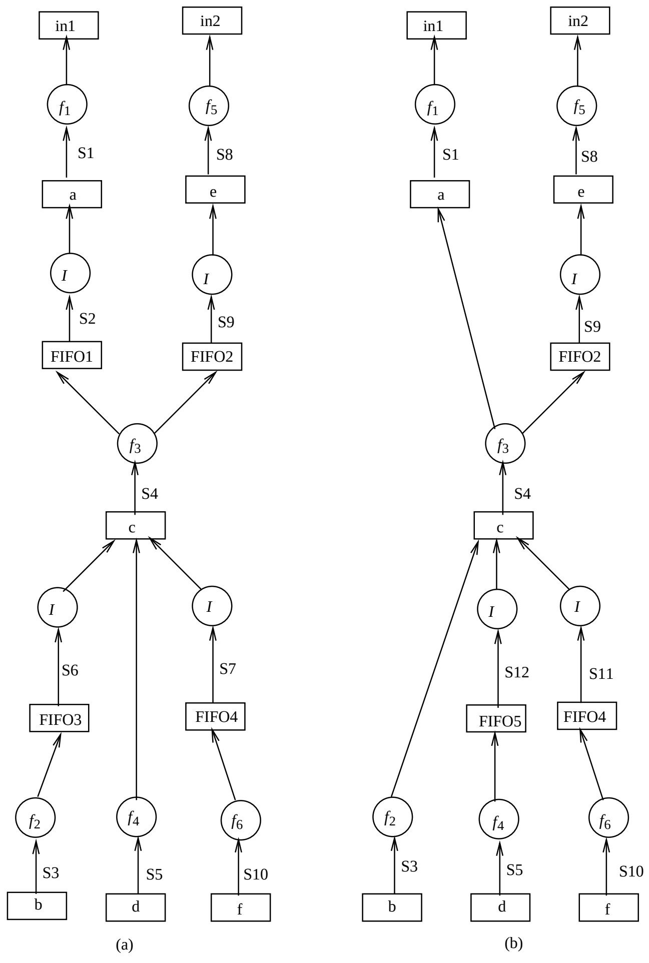


Figure 6.36: Computation Migration: (a) ADDG of the initial KPN; (b) ADDG of the transformed KPN

6.6 Experimental results

The methodology described in this note has been implemented and run on a 2.0 GHz Intel® Core™2 Duo machine. We have integrated the SMT solver Yices (Yices, 2010) in our tool to find the relation between fIn ($fOut$) and the loop indices. For obtaining the dependence mappings of the slices, our tool relies on the Omega calculator (Kelly et al., 2008). The method has been tested on the sequential behaviours of example 21 (PRG), the Sobel edge detection (SOB), Daubechies 4-coefficient wavelet filter (WAVE) and Laplace algorithm for edge enhancement of northerly directional edges (LAP). The KPNs are generated from the sequential behaviours using the method given in (Turjan, 2007). The number of processes and FIFOs in the KPN, the numbers of arrays, slices and slice classes in the sequential behaviours and its corresponding

	KPN		*array*		*slice*		*slice class*		Exec
Benchmark	*process*	*fifo*	*seq*	*kpn*	*seq*	*kpn*	*seq*	*kpn*	*(sec)*
PRG	4	4	6	10	2	6	2	2	03.49
SOB	5	14	4	18	1	1	1	1	11.30
WAVE	5	16	2	18	4	4	2	2	09.22
LAP	2	9	2	11	1	1	1	1	06.17

Table 6.1: Results for sequential to KPN transformation

KPN behaviours and the execution times are tabulated for each of the test cases in table 6.1. In all the cases, our method was able to establish the equivalence in less than twelve seconds.

	processes		*FIFOs*		*slices*		time
Benchmark	*src*	*trans*	*src*	*trans*	*src*	*trans*	*(sec)*
Channel merging	2	2	2	1	4	8	0.601
Channel splitting	2	2	1	2	2	1	0.398
Process splitting	2	3	1	2	2	4	0.367
Process merging	4	3	4	4	3	3	0.500
Computation migration	3	3	4	3	3	3	1.204

Table 6.2: Results for KPN level transformations

To test the KPN level transformations, we consider examples provided in this chapter. The number of processes, the number of FIFOs, the number of slices and the execution time of our method are tabulated in table 6.2. We could not run the example of message vectorization since the present implementation of our method does not support message de-vectorization. For all other cases, our method succeeds in establishing the equivalence in less than two seconds.

6.7 Conclusion

This chapter presents an equivalence checking method for verification of sequential to parallel KPN code transformations often used in multimedia and signal processing applications. The problem is suitably mapped to the equivalence problem of ADDGs. The FIFO communication among the KPN processes has necessitated non-trivial enhancement of the basic ADDG construction method from sequential programs. The individual KPN processes are sequential; application of the basic ADDG construction mechanism on each of them, however, necessitates that FIFO communication be properly captured. The method visualizes the FIFOs as linear arrays, each with two pointers, one for the consumer and the other for the producer. Significant processing is needed to obtain the relationship of these pointers with the loop iterators; the association can be non-linear even for simple cases. The method then composes the individual ADDGs into a single ADDG around the FIFOs. It has formally established that as long as the association of the FIFO pointers with loop iterators are linear and there is no deadlock, the composed ADDG captures the dependencies across all KPN processes. It has, however, been identified that while deadlocks due to circular dependencies among the processes get accounted for automatically in the composition process, the deadlock due to insufficient communication of data eludes the mechanism; an enhancement is presented to take case of such cases. The effectiveness of the mechanism for verification of various optimizing transformations on KPN behaviours are also presented. The experimental results on several test cases demonstrate the effectiveness of our method.

Chapter 7

Conclusion and Future Scopes

The formal verification of embedded systems is an emerging area of research requiring several verification problems, with different objectives, to be addressed. Verification of behavioural transformations that are applied during embedded system design is one of the major problems from the gamut of embedded system verification problems. We have addressed following six such behavioural transformation verification problems in this thesis:

1. Verification of code motion techniques
2. Verification of RTL generation phase in high-level synthesis
3. Verification of RTL transformations
4. Verification of loop and arithmetic transformations of array intensive behaviours
5. Verification of sequential to KPN code transformations
6. Verification of KPN level transformations.

In this chapter, we first summarize the contributions of this thesis. We then discuss some directions of future research.

7.1 Summary of contributions

Verification of code motion transformations: An equivalence checking method for checking correctness of both uniform and non-uniform code motion techniques is presented in Chapter 3. Both the input and the output behaviours are represented as FSMDs. Our method introduces cutpoints in one FSMD, visualizes its computations as concatenation of paths from cutpoints to cutpoints, and then identifies equivalent finite path segments in the other FSMD; the process is then repeated with the FSMDs interchanged. A path is extended when its equivalent path cannot be found in the other FSMD. However, a path cannot be extended beyond loop boundaries. It is shown that blind path extension does not work for non-uniform cases of code motions. We have underlined that for non-uniform code motions, identifying equivalent path segments is effectively handled by model checking of some data-flow related properties. Specifically, if the property *'always defined before being used'* is true for a variable v at the end state of a path, then the value of v at that state can be ignored for checking equivalence of the rest of the behaviour since that particular value of v has no further use. Accordingly, a path will be extended only when its equivalent path cannot be found due to mismatch of value of a variable v in the other FSMD and the property *'always defined before being used'* is false for v at the end state of the path. The property is encoded as the CTL formula $\mathrm{E}[(\neg d_v)\ \mathrm{U}\ u_v]$ and the FSMD is suitably converted to an equivalent Kripke structure. Our method automatically identifies situations where such properties need to be checked during equivalence checking, generates the appropriate properties and invokes the model checking tool NuSMV to verify them. The method also applies normalization of arithmetic expressions over integer and some simplification rules to handle arithmetic transformations. The method has been proved to be sound and always terminating. The complexity of the method is exponential in the worst case and polynomial in the best case on the number of states in the FSMDs. We have implemented our method and have tested on the results of the scheduling steps of the HLS tool SPARK which applies a wide range of code motion techniques during the scheduling process. It is shown through experimentation that our method is capable of handling code motions applied by SPARK and the worst case time complexity situation is usually not encountered for the practical cases of equivalence checking.

Verification of RTL generation phase in high-level synthesis: A verification framework for the RTL generation phase of high-level synthesis is presented in Chapter 4. The input of this phase is a scheduled behaviour and the output is an RTL design which consists of a description of the datapath netlist and a controller FSM. The input behaviour can be modelled as an FSMD. In this work, we develop a method to represent the RTL designs generated through high level synthesis as FSMDs. The method is strong enough to handle pipelined and multicycle operations, if any, spanning over several states. The goal is achieved in two steps. In the first step, the datapath interconnection and the controller FSM description of the RTL design generated by a high-level synthesis procedure are analyzed to obtain the register transfer (RT) operations executed in the datapath for a given control assertion pattern in each control step. In the second step, the FSMD based equivalence checking method developed for the previous problem is deployed to establish equivalence between the input and the output behaviours of this phase. For a multicycle operation, our method automatically checks in the first step (i) whether the controller generates proper control signals so that the data are held constant on the FU inputs over all the control steps it takes to execute that operation and (ii) the registers feeding the inputs to the FU are not updated (by any other operations) during execution of the multicycle operation. For a k-stage pipelined operation scheduled in the i^{th} time step, our method automatically checks whether the datapath from the source registers to the FU inputs are set at the i^{th} time step and the datapath from the output of the FU to the destination register is set at the $(i+k-1)^{th}$ time step by the controller. An FSMD is abstracted out from the RTL design by performing the above tasks for each state of the controller FSM. We have shown that several inconsistencies both in the datapath and in the controller can be detected during construction of the FSMD from the RTL. It has been shown that the application of normalization techniques to represent arithmetic expressions over integers during equivalence checking of FSMDs helps us handle algebraic transformation techniques based on commutativity, associativity and distributivity of arithmetic operations that are often used during datapath synthesis to improve interconnection cost. The FSMD construction mechanism is proven to be sound and complete. The complexity of the construction method is shown to be polynomial in terms of the datapath and controller sizes. The method is implemented and integrated with an existing HLS tool, called SAST. Experimental results supporting this scheme are provided.

Verification of RTL transformations: A verification framework for RTL transformations are also presented in Chapter 4. Specifically, we analyze commonly applied low power RTL transformation techniques such as, restructuring multiplexer networks (to enhance data correlations and eliminate glitchy control signals), clocking control signals, and inserting selective rising/falling delays, etc., and show that the number of control signals and the datapath interconnections may change due to these transformations. We then show that our rewriting mechanism by which the concurrent register transfer operations are identified can handle these transformations.

Verification of loop and arithmetic transformations of array intensive behaviours: In Chapter 5, we propose a formal verification method for checking correctness of loop transformations and arithmetic transformations applied on the array and loop intensive behaviours in design of area/energy efficient systems in the domain of multimedia and signal processing applications. Loop transformations hinge more on data dependence and index space of the arrays than on the control flow of the behaviour. Hence, array data dependence graphs (ADDGs), proposed by Shashidhar (Shashidhar, 2008), are used for representing array intensive behaviours. They proposed an ADDG based equivalence checking method to validate the loop transformations. Possible enhancements of his method to handle associative and commutative transformations are also discussed in (Shashidhar, 2008; Shashidhar et al., 2005a). In this work, we redefine the equivalence of ADDGs based on a slice level characterization of ADDGs to verify loop transformations along with a wide range of arithmetic transformations. Specifically, we introduce the notion of data transformation of a slice. A slice is characterized based on the data transformation and the dependence mappings involved in the slice. Arithmetic transformation of expressions can involve arrays; in such situations, the ordering of terms in the normalized expression needs to be carried out using the order of the normalized dependence mappings of the several occurrences of the arrays in the terms. Accordingly, a normalization method is proposed for integer arithmetic expressions involving array references which uniformly represents both the data transformation and the dependence mappings of a slice. We also propose some simplification rules of normalized expressions to handle arithmetic transformations. The equivalence checking method proposed in this work relies on this normalization technique and the simplification rules to handle arithmetic transformations over arrays. Correctness and complexity of the method have been dealt with. Experimental results supporting this scheme are provided.

Verification of sequential to parallel code transformations: A formal verification method for checking correctness of sequential to KPN transformations is presented in Chapter 6. The idea is to model both the initial sequential behaviour and the output KPN behaviour as ADDGs and then apply our ADDG based equivalence checker to establish the equivalence. We describe a method to represent a KPN behaviour comprising inherently parallel processes as an ADDG. The basic steps involved in modelling a KPN as an ADDG are as follows: (i) each process of the KPN is first modelled as an ADDG and (ii) the ADDGs corresponding to the processes are then composed to obtain the ADDG of the KPN. In the first step, each FIFO channel in a process of a KPN is first replaced by a linear array associated with two indices, one each for handling the primitive operations *get* and *put* on FIFO. In order to encode these modified sequential behaviours (corresponding to the processes) by ADDGs, we need to identify the dependence mappings of the FIFO pointers with the indices of the loop where a FIFO is used/defined. The ADDGs of two processes are composed when they are communicated through a FIFO. To compute the dependence mappings across a FIFO, we use the *first-if-first-out* property of FIFO. The correctness of the ADDG composition step is proven for a deadlock free KPN. The presence of deadlock, however, is found to impair the correctness of composition. Two kinds of deadlock situations have been identified, one of which is accounted for during composition whereas the other needs extra processing step which has been incorporated. Experimental results supporting this scheme are provided.

Verification of parallel code transformations: We have also addressed the problem of verifying KPN level transformations in Chapter 6. We model both the input and the transformed KPNs as ADDGs and apply our ADDG based equivalence method for establishing the equivalence between the two KPNs. Specifically, we consider commonly applied KPN transformations such as, channel merging, channel splitting, process merging, process splitting, message vectorization and computation migration, etc. and show that the number of FIFOs and the number of processes in a KPN, the level of parallelism in the KPN and the communication order in the FIFO may be modified due to these transformations. We then show how these transformations can be verified in our verification framework. Experimental results supporting this scheme are provided.

7.2 Scope for future work

The methods developed in this work can be enhanced to overcome their limitations. Also, there is scope of application of the developed methods in other verification problems. In the following, we discuss both aspects of future works.

7.2.1 Enhancement of the present work

Verification of code motion across loop boundaries: The inherent limitation of our path extension based equivalence of FSMDs is that a path cannot be extended beyond loop boundaries as that is prevented by the definition of a path cover. As a result, the method fails when code segment is moved beyond a loop boundary. In situations where code segments are moved completely across loops, our method results in a number of extensions of paths before being failing encountering a loop barrier. One solution can be to propagate the mismatched variable values to all the subsequent path segments. One immediate objective can be consolidating this approach of value propagation to enhance our equivalence checking method to remedy the aforesaid limitation of our path extension based method.

Combining ADDG and FSMD based methods: The FSMD based equivalence checking method is developed with the intention of verification of code motion techniques and control structure modification transformations primarily in control intensive behaviours. The ADDG based equivalence checking method, on the other hand, is developed to verify loop and arithmetic transformations of data intensive behaviours. However, this model does not support data dependent conditions. Interestingly, the FSMD based modelling supports data dependent conditions but it does not support most of the loop transformations. A combination of the power of these two models, therefore, appears to be promising to realize a more powerful equivalence checker.

7.2.2 Scope of application to other research areas

Evolving programs: Software is not written entirely from scratch. In general, a software is gradually evolved over time. In industrial software development projects, this complexity of software evolution is explicitly managed via check-in of program versions. Validation of such evolving programs remains a huge problem in terms of software development (Qi et al., 2009). An interesting study would be to check the applicability of the formal methods developed in this work to establish equivalence of evolving programs.

Automatic code generation: Automatic code generation is an enabling technology for model based software development and has significant potential to improve the entire software development process. While model based automatic code generation tools exist, they do not offer any verification guarantees for the generated code. When embedded software is employed in safety-critical applications such as, automobiles, autonomous medical devices, etc., the need of high assurance is obvious. The application of formal method into verification of code generation process is an interesting research challenge which introduces rigor into the code generation step. Testing based approaches for verifying auto-code generators exist (Majumdar and Xu, 2007; Sampath et al., 2008). Therefore, identifying the scope of applications of the equivalence checking methods developed in this work in verification of code generation process can have a significant impact on reliability of control software.

Automatic program evaluation: All large educational institutions or universities across the world, specifically in India, have a very large cohorts of students where the intake of undergraduates is around 1000 (Mandal et al., 2007). As a part of their curriculum, the students need to attend laboratories and each student has to submit about 9 to 12 assignments and up to 3 laboratory based tests. That amounts to nearly 15,000 submissions per semester. Due to increase in number of seats in the government institutions, the situation will become ever worse. Without automation, the instructor would be busy most of the time in testing and grading work at the expense of time that could be spent interacting with students. A formal automatic evaluation system would add a significant value to the education system. The evaluation tool assists instructors by automatically evaluating, marking, and providing critical feedback for programming assignments. It would be an interesting study to check how our FSMD based equivalence checking method whereupon each student program is analyzed with respect to a model program can be enhanced for these purposes.

Application to architecture mapping: The processes of the KPN are finally mapped to processors and the FIFO channels are mapped to local or shared memory. Several properties, such as "system will never reach a deadlock" or "the system will satisfy all the real time timing constraints", etc., need to be verified for which model checking may be a better choice for this phase compared to equivalence checking. However, the final systems may be viewed as globally asynchronous locally synchronous (GALS) systems on which the application tasks are performed in the spatial domain and communication among the component modules is achieved by message passing. So, it is possible to construct an abstract model of the overall system and then show the equivalence with the KPN behavioural model. This task is a nontrivial one and the scope of equivalence checking in this phase is worth examining.

7.3 Conclusion

Embedded system design flow comprises various phases where each phase performs some specific tasks algorithmically providing for ingenious intervention of experts. The gap between the original behavior and the finally synthesized circuit is too wide to be analyzed by any monolithic reasoning mechanism. The validation tasks, therefore, must be planned to go hand in hand with each phase of the design flow. This thesis addressed verification of certain behavioural transformation phases in the embedded system design flow. We believe integrating these methods with embedded system design tools will increase the dependability of embedded systems.

Bio-data

Chandan Karfa was born in Shyamsundar, Burdwan, West Bengal on 9th of May, 1982. He received the B.Tech. degree in Information Technology from University Science Instrumentation Center of University of Kalyani in 2004 and the M.S. (by research) degree in Computer Science and Engineering from Indian Institute of Technology, Kharagpur in 2007. He has worked as a Junior Project Assistance (JPA) in sponsored project from July 2004 to February 2007 and as a Research Consultant from March 2007 to August 2008 in the VLSI Consortium project undertaken by the Advanced VLSI Design Laboratory, IIT Kharagpur. His current research interests include formal verification, electronic design automation and embedded system verification. He has published twelve research papers in different reputed IEEE/ACM international journals and conferences. He has received Innovative Student Projects Award from Indian National Academy of Engineers in 2008, Best Student Paper Award in ADCOM conference in 2007, Microsoft Research India PhD Fellowship in 2008 and First prize in EDA contest in VLSI Design conference in 2009.

List of Publications / Communications out of this work

1. **C. Karfa**, C. Mandal, D. Sarkar; *Formal Verification of Code Motion Techniques using Data-flow Driven Equivalence Checking*; ACM Transactions on Design Automation of Electronic Systems (TODAES). (under revision)

2. **C. Karfa**, D. Sarkar, C. Mandal; *Verification of Datapath and Controller Generation Phase in High-level Synthesis of Digital Circuits*; IEEE Transactions on COMPUTER-AIDED DESIGN of Integrated Circuits and Systems (TCAD); page 479-492, Vol. 29, No. 3, March, 2010.

3. **C. Karfa**, D. Sarkar, C. Mandal; *An Equivalence Checking Method for Scheduling Verification in High-level Synthesis*; IEEE Transactions on COMPUTER-AIDED DESIGN of Integrated Circuits and Systems (TCAD); page 556-569, Vol 27, No. 3, March, 2008.

4. **C. Karfa**, D. Sarkar, C. Mandal; *Verification of Register Transfer Level Low Power Transformations*; IEEE Computer Society Annual Symposium on VLSI (ISVLSI) 2011; page 313-314, July 4-6, Chennai, India, 2011.

5. **C. Karfa**, K. Banerjee, D. Sarkar, C. Mandal; Equivalence Checking of Array-Intensive Programs; IEEE Computer Society Annual Symposium on VLSI (ISVLSI); page 156-161, July 4-6, Chennai, India, 2011.

6. **C. Karfa**, D. Sarkar, C. Mandal; *Data-flow Driven Equivalence Checking for Verification of Code Motion Techniques*; IEEE Computer Society Annual Symposium on VLSI (ISVLSI) 2010; page 428-433, July 5-7, Lixouri Kefalonia, Greece, 2010.

Bibliography

Aho, A. V., Sethi, R. and Ullman, J. D. (1987). COMPILERS Principles, Techniques and Tools, Addison-Wesley Publishing Company.

Ahuja, S., Zhang, W., Lakshminarayana, A. and Shukla, S. K. (2010). A Methodology for Power Aware High-Level Synthesis of Co-processors from Software Algorithms, in Proc. of the VLSID '10, pp. 282–287, ISBN 978-0-7695-3928-7.

Ashar, P., Bhattacharya, S., Raghunathan, A. and Mukaiyama, A. (1998). Verification of RTL generated from scheduled behavior in a high-level synthesis flow, in Proc. of the 1998 IEEE/ACM international conference on Computer-aided design, ICCAD '98, pp. 517–524, ACM, ISBN 1-58113-008-2.

Bacon, D. F., Graham, S. L. and Sharp, O. J. (1994). Compiler transformations for high-performance computing, ACM Comput. Surv., vol. 26, pp. 345–420.

Barrett, C. W., Fang, Y., Goldberg, B., Hu, Y., Pnueli, A. and Zuck, L. D. (2005). TVOC: A Translation Validator for Optimizing Compilers, in CAV, pp. 291–295.

Bergamaschi, R. A., Lobo, D. and Kuehlmann, A. (1992). Control optimization in high-level synthesis using behavioral don't cares, in Proc. of the DAC, pp. 657–661, ISBN 0-89791-516-X.

Bharath, N., Nandy, S. and Bussa, N. (2005). Artificial Deadlock Detection in Process Networks for ECLIPSE, in IEEE International Conference on Application-Specific Systems, Architecture Processors 2005, pp. 22–27, ISBN 0-7695-2407-9.

Borrione, D., Dushina, J. and Pierre, L. (2000). A compositional model for the functional verification of high-level synthesis results, IEEE Transactions on VLSI Systems, vol. 8, no. 5, pp. 526–530.

Bouchebaba, Y., Girodias, B., Nicolescu, G., Aboulhamid, E. M., Lavigueur, B. and Paulin, P. (2007). MPSoC memory optimization using program transformation, ACM Trans. Des. Autom. Electron. Syst., vol. 12, no. 4, pp. 43:1–43:39.

Brandolese, C., Fornaciari, W., Salice, F. and Sciuto, D. (2004). Analysis and Modeling of Energy Reducing Source Code Transformations, in Proc. of the conference on Design, automation and test in Europe, DATE ’04, pp. 306–311, IEEE Computer Society, Washington, DC, USA, ISBN 0-7695-2085-5.

Buck, J. T. (1993). Scheduling Dynamic Dataflow Graphs with Bounded Memory Using the Token Flow Model, Ph.D. thesis, University of California, EECS Dept., Berkeley, CA.

Camposano, R. (1991). Path-Based Scheduling for Synthesis, IEEE transactions on computer-Aided Design of Integrated Circuits and Systems, vol. Vol 10 No 1, pp. 85–93.

Carpenter, P. M., Ramirez, A. and Ayguade, E. (2010). Buffer sizing for self-timed stream programs on heterogeneous distributed memory multiprocessors, in 5th International Conference on High Performance Embedded Architectures and Compilers, HiPEAC 2010, pp. 96–110, Springer.

Castrillon, J., Velasquez, R., Stulova, A., Sheng, W., Ceng, J., Leupers, R., Ascheid, G. and Meyr, H. (2010). Trace-based KPN composability analysis for mapping simultaneous applications to MPSoC platforms, in Proceedings of the Conference on Design, Automation and Test in Europe, DATE ’10, pp. 753–758, European Design and Automation Association, 3001 Leuven, Belgium, Belgium, ISBN 978-3-9810801-6-2.

Catthoor, F., D., E. and Greff, S. S. (1998). HICSS. Custom Memory Management Methodology:Exploration of Memory Organisation for Embedded Multimedia System Design, Kluwer Academic Publishers.

Ceng, J., Castrillon, J., Sheng, W., Scharwächter, H., Leupers, R., Ascheid, G., Meyr, H., Isshiki, T. and Kunieda, H. (2008). MAPS: an integrated framework for MPSoC application parallelization, in Proceedings of the 45th annual Design Automation Conference, DAC ’08, pp. 754–759, ACM, New York, NY, USA, ISBN 978-1-60558-115-6.

Chandrakasan, A., Potkonjak, M., Mehra, R., Rabaey, J. and Brodersen, R. (1995a). Optimizing power using transformations, IEEE Transactions on CAD of ICS, vol. 14, no. 1, pp. 12 –31.

Chandrakasan, A., Sheng, S. and Brodersen, R. (1992). Low-power CMOS digital design, IEEE Journal of Solid-State Circuits, vol. 27, no. 4, pp. 473 –484.

Chandrakasan, A. P., Potkonjak, M., Mehra, R., Rabaey, J. and Brodersen, R. W. (1995b). Optimizing Power Using Transformations, IEEE Transactions on CAD of ICS, vol. 14, no. 1, pp. 12–31.

Chen, G., Kandemir, M. and Li, F. (2006a). Energy-aware computation duplication for improving reliability in embedded chip multiprocessors, in ASP-DAC ’06: Proceedings of the 2006 conference on Asia South Pacific design automation, pp. 134–139, IEEE Press, Piscataway, NJ, USA, ISBN 0-7803-9451-8.

Chen, X., Hsieh, H. and Balarin, F. (2006b). Verification approach of metropolis design framework for embedded systems, Int. J. Parallel Program., vol. 34, pp. 3–27.

Chen, X., Hsieh, H., Balarin, F. and Watanabe, Y. (2003). Formal Verification for Embedded System Designs, Design Automation for Embedded Systems, vol. 8, pp. 139–153.

Cheung, E., Chen, X., Hsieh, H., Davare, A., Sangiovanni-Vincentelli, A. and Watanabe, Y. (2009). Runtime deadlock analysis for system level design, Design Automation for Embedded Systems, vol. 13, pp. 287–310.

Cheung, E., Hsieh, H. and Balarin, F. (2007). Automatic buffer sizing for rate-constrained KPN applications on multiprocessor system-on-chip, in Proceedings of the 2007 IEEE International High Level Design Validation and Test Workshop, pp. 37–44, IEEE Computer Society, Washington, DC, USA, ISBN 978-1-4244-1480-2.

Chiang, T.-H. and Dung, L.-R. (2007). Verification method of dataflow algorithms in high-level synthesis, J. Syst. Softw., vol. 80, no. 8, pp. 1256–1270.

Cimatti, A., Clarke, E. M., Giunchiglia, F. and Roveri, M. (2000). NuSMV: A New Symbolic Model Checker, International Journal on Software Tools for Technology Transfer, vol. 2, no. 4, pp. 410–425.

Clarke, E. M., Grumberg, O. and Peled, D. A. (2002). Model Checking, The MIT Press.

Cockx, J., Denolf, K., Vanhoof, B. and Stahl, R. (2007). SPRINT: a tool to generate concurrent transaction-level models from sequential code, EURASIP J. Appl. Signal Process., vol. 2007, no. 1, pp. 1–15.

Cordone, R., Ferrandi, F., Santambrogio, M. D., Palermo, G. and Sciuto, D. (2006). Using speculative computation and parallelizing techniques to improve scheduling of control based designs, in Proceedings of the 2006 Asia and South Pacific Design Automation Conference, ASP-DAC ’06, pp. 898–904, IEEE Press, Piscataway, NJ, USA, ISBN 0-7803-9451-8.

Cormen, T. H., Leiserson, C. E., Rivest, R. L. and Stein, C. (2001). Introduction to Algorithms, The MIT Press, Cambridge , Massachusetts London, England.

Davare, A., Zhu, Q. and Sangiovanni-Vincentelli, A. (2006). A Platform-based Design Flow for Kahn Process Networks, Tech. Rep. 2006-30, UC Berkeley.

de Kock, E. A. (2002). Multiprocessor mapping of process networks: a JPEG decoding case study, in ISSS ’02: Proceedings of the 15th international symposium on System Synthesis, pp. 68–73, ISBN 1-58113-576-9.

Dos Santos, L. C. V., Heijligers, M. J. M., Van Eijk, C. A. J., Van Eijnhoven, J. and Jess, J. A. G. (2000). A code-motion pruning technique for global scheduling, ACM Trans. Des. Autom. Electron. Syst., vol. 5, no. 1, pp. 1–38.

Dos. Santos, L. C. V. and Jress, J. (1999). A reordering technique for efficient code motion, in Procs. of the 36th ACM/IEEE Design Automation Conference, DAC ’99, pp. 296–299, ACM, New York, NY, USA, ISBN 1-58113-109-7.

E. Özer, A. P. N. and Gregg, D. (2003). Classification of Compiler Optimizations for High Performance, Small Area and Low Power in FPGAs, Tech. rep., Trinity College, Dublin, Ireland, Department of Computer Science.

Eveking, H., Hinrichsen, H. and Ritter, G. (1999). Automatic verification of scheduling results in high-level synthesis, in Proceedings of the conference on Design, automation and test in Europe, DATE ’99, pp. 260–265, ACM, New York, NY, USA, ISBN 1-58113-121-6.

Fehnker, A., Huuck, R., Jayet, P., Lussenburg, M. and Rauch, F. (2007). Model Checking Software at Compile Time, in Proceedings of the First Joint IEEE/IFIP Symposium on Theoretical Aspects of Software Engineering, pp. 45–56, IEEE Computer Society, Washington, DC, USA, ISBN 0-7695-2856-2.

Fehnker, A., Huuck, R., Schlich, B. and Tapp, M. (2009). Automatic Bug Detection in Microcontroller Software by Static Program Analysis, in Proceedings of the 35th Conference on Current Trends in Theory and Practice of Computer Science, SOFSEM ’09, pp. 267–278, Springer-Verlag, Berlin, Heidelberg, ISBN 978-3-540-95890-1.

Fei, Y., Ravi, S., Raghunathan, A. and Jha, N. K. (2007). Energy-optimizing source code transformations for operating system-driven embedded software, Trans. on Embedded Computing Sys., vol. 7, no. 1, pp. 1–26.

Ferrandi, F., Fossati, L., Lattuada, M., Palermo, G., Sciuto, D. and Tumeo, A. (2007). Automatic Parallelization of Sequential Specifications for Symmetric MPSoCs, IFIP International Federation for Information Processing, vol. 231, no. 2, pp. 179–192.

Fisher, J. (1981). Trace Scheduling: A Technique for Global Microcode Compaction, IEEE Transactions on Computers, vol. C-30, no. 7, pp. 478 –490.

Floyd, R. W. (1967). Assigning Meaning to programs, in Schwartz, J. T. (Ed.), Proceedings the 19th Symposium on Applied Mathematics, pp. 19–32, American Mathematical Society, Providence, R.I., mathematical Aspects of Computer Science.

Fraboulet, A., Kodary, K. and Mignotte, A. (2001). Loop fusion for memory space optimization, in ISSS ’01: Proceedings of the 14th international symposium on Systems synthesis, pp. 95–100, ISBN 1-58113-418-5.

Freisleben, B. and Kielmann, T. (1995). Automated Transformation of Sequential Divide-and-Conquer Algorithms into Parallel Programs, Computers and Artificial Intelligence, vol. 14, pp. 579–596.

Fujita, M. (2005). Equivalence checking between behavioral and RTL descriptions with virtual controllers and datapaths, ACM Trans. Des. Autom. Electron. Syst., vol. 10, no. 4, pp. 610–626.

Gajski, D. D., Dutt, N. D., Wu, A. C. and Lin, S. Y. (1992). High-Level Synthesis: Introduction to Chip and System Design, Kluwer Academic Publishers.

Geilen, M. and Basten, T. (2003). Requirements on the execution of Kahn process networks, in Proceedings of the 12th European conference on Programming, ESOP'03, pp. 319–334, Springer-Verlag, Berlin, Heidelberg, ISBN 3-540-00886-1.

Gesellensetter, L., Glesner, S. and Salecker, E. (2008). Formal verification with Isabelle/HOL in practice: finding a bug in the GCC scheduler, in Proceedings of the 12th international conference on Formal methods for industrial critical systems, FMICS'07, pp. 85–100, Springer-Verlag, Berlin, Heidelberg, ISBN 3-540-79706-8, 978-3-540-79706-7.

Ghodrat, M., Givargis, T. and Nicolau, A. (2008). Control flow optimization in loops using interval analysis, in Proceedings of the 2008 international conference on Compilers, architectures and synthesis for embedded systems, CASES '08, pp. 157–166, ACM, New York, NY, USA, ISBN 978-1-60558-469-0.

Ghodrat, M., Givargis, T. and Nicolau, A. (2009). Optimizing control flow in loops using interval and dependence analysis, Design Automation for Embedded Systems, vol. 13, pp. 193–221.

Girkar, M. and Polychronopoulos, C. D. (1992). Automatic Extraction of Functional Parallelism from Ordinary Programs, IEEE Trans. Parallel Distrib. Syst., vol. 3, no. 2, pp. 166–178.

Graphics, M. (2006). Catapult C Synthesis.
http://www.mentor.com/products/esl/high_level_synthesis/

Gries, D. (1987). The Science of Programming, Springer-Verlag New York, Inc., Secaucus, NJ, USA, ISBN 0387964800.

Gupta, R. and Soffa, M. (1990). Region scheduling: an approach for detecting and redistributing parallelism, IEEE Transactions on Software Engineering, vol. 16, no. 4, pp. 421 –431.

Gupta, S., Dutt, N., Gupta, R. and Nicolau, A. (2003a). Dynamic Conditional Branch Balancing during the High-Level Synthesis of Control-Intensive Designs, in Proceedings of DATE'03, pp. 270–275, IEEE Computer Society, Washington, DC, USA, ISBN 0-7695-1870-2.

Gupta, S., Dutt, N., Gupta, R. and Nicolau, A. (2003b). Dynamically Increasing the Scope of Code Motions during the High-Level Synthesis of Digital Circuits, IEE Proceedings: Computer and Digital Technique, vol. 150, no. 5, pp. 330–337.

Gupta, S., Dutt, N., Gupta, R. and Nicolau, A. (2003c). SPARK: a high-level synthesis framework for applying parallelizing compiler transformations, in Proc. of Int. Conf. on VLSI Design, pp. 461–466, IEEE Computer Society, Washington, DC, USA.

Gupta, S., Dutt, N., Gupta, R. and Nicolau, A. (2004a). Loop Shifting and Compaction for the High-level Synthesis of Designs with Complex Control Flow, in Proceedings of the DATE ’04, vol. 1, pp. 114–119.

Gupta, S., Dutt, N., Gupta, R. and Nicolau, A. (2004b). Using global code motions to improve the quality of results for high-level synthesis, IEEE Transactions on CAD of ICS, vol. 23, no. 2, pp. 302–312.

Gupta, S., Gupta, R., Dutt, N. and Nicolau, A. (2004c). Coordinated Parallelizing Compiler Optimizations and High-Level Synthesis, ACM Transactions on Design Automation of Electronic Systems (TODAES), vol. 9, no. 4, pp. 1–31.

Gupta, S., Miranda, M., Catthoor, F. and Gupta, R. (2000). Analysis of high-level address code transformations for programmable processors, in Proceedings of the conference on Design, automation and test in Europe, DATE ’00, pp. 9–13, ACM, New York, NY, USA, ISBN 1-58113-244-1.

Gupta, S., Reshadi, M., Savoiu, N., Dutt, N., Gupta, R. and Nicolau, A. (2002). Dynamic common sub-expression elimination during scheduling in high-level synthesis, in Proceedings of the 15th international symposium on System Synthesis, ISSS ’02, pp. 261–266, ACM, New York, NY, USA, ISBN 1-58113-576-9.

Gupta, S., Savoiu, N., Dutt, N., Gupta, R. and Nicolau, A. (2001a). Conditional Speculation and its Effects on Performance and Area for High-Level Synthesis, in International Symposium on System Synthesis, pp. 171–176.

Gupta, S., Savoiu, N., Kim, S., Dutt, N., Gupta, R. and Nicolau, A. (2001b). Speculation techniques for high level synthesis of control intensive designs, in Proceedings of DAC’01, pp. 269–272.

Hall, M. W., Anderson, J. M., Amarasinghe, S. P., Murphy, B. R., Liao, S.-W., Bugnion, E. and Lam, M. S. (1996). Maximizing Multiprocessor Performance with the SUIF Compiler, Computer, vol. 29, no. 12, pp. 84–89.

Hoare, C. A. R. (1969). An axiomatic basis for computer programming, Commun. ACM, vol. 12, pp. 576–580.

Holzmann, G. J. (1997). The Model Checker SPIN, IEEE Trans. Softw. Eng., vol. 23, pp. 279–295.

Howden, W. E. (1987). Functional program testing and analysis, McGraw-Hill, New York.

Hu, Y., Barrett, C., Goldberg, B. and Pnueli, A. (2005). Validating More Loop Optimizations, Electronic Notes in Theoretical Computer Science, vol. 141, no. 2, pp. 69–84, proceedings of the Fourth International Workshop on Compiler Optimization meets Compiler Verification (COCV 2005).

Hwu, W. M. W., Mahlke, S. A., Chen, W. Y., Chang, P. P., Warter, N. J., Bringmann, R. A., Ouellette, R. G., Hank, R. E., Kiyohara, T., Haab, G. E., Holm, J. G. and Lavery, D. M. (1993). The superblock: An effective technique for VLIW and superscalar compilation, The Journal of Supercomputing, vol. 7, pp. 229–248.

Jain, R., Majumdar, A., Sharma, A. and Wang, H. (1991). Empirical evaluation of some high-level synthesis scheduling heuristics, in Proceedings of the 28th ACM/IEEE Design Automation Conference, DAC ’91, pp. 686–689, ACM, New York, NY, USA, ISBN 0-89791-395-7.

Jiang, B., Deprettere, E. and Kienhuis, B. (2008). Hierarchical run time deadlock detection in process networks, in IEEE Workshop on Signal Processing Systems, 2008, pp. 239 –244.

Jiong, L., Lin, Z., Yunsi, F. and Jha, N. (2004). Register binding-based RTL power management for control-flow intensive designs, IEEE Transactions on CAD of ICS, vol. 23, no. 8, pp. 1175 – 1183.

Johnson, N. E. (2004). Code Size Optimization for Embedded Processors, Ph.D. thesis, University of Cambridge.

Kadayif, I. and Kandemir, M. (2005). Data space-oriented tiling for enhancing locality, Trans. on Embedded Computing Sys., vol. 4, no. 2, pp. 388–414.

Kahn, G. (1974). The Semantics of a Simple language for Parallel Programming, in Proceedings of IFIP Congress, pp. 471–475, North Holland Publishing Company.

Kandemir, M., Son, S. W. and Chen, G. (2005). An evaluation of code and data optimizations in the context of disk power reduction, in ISLPED '05: Proceedings of the 2005 international symposium on Low power electronics and design, pp. 209–214, ISBN 1-59593-137-6.

Kandemir, M., Vijaykrishnan, N., Irwin, M. J. and Ye, W. (2001). Influence of compiler optimizations on system power, IEEE Trans. Very Large Scale Integr. Syst., vol. 9, pp. 801–804.

Kandemir, M. T. (2006). Reducing energy consumption of multiprocessor SoC architectures by exploiting memory bank locality, ACM Trans. Des. Autom. Electron. Syst., vol. 11, no. 2, pp. 410–441.

Karakoy, M. (2005). Optimizing Array-Intensive Applications for On-Chip Multiprocessors, IEEE Trans. Parallel Distrib. Syst., vol. 16, no. 5, pp. 396–411.

Karfa, C. (2007). Hand-in-hand Verification and Synthesis of Digital Circuits, MS Thesis, IIT Kharagpur.

Karfa, C., Reddy, J., Mandal, C. R., Sarkar, D. and Biswas, S. (2005). SAST: An interconnection aware high-level synthesis tool, in Proc. 9th VLSI Design and Test Symposium, Bangalore, pp. 285–292.

Karp, R. M., Miller, R. E. and Winograd, S. (1967). The Organization of Computations for Uniform Recurrence Equations, J. ACM, vol. 14, pp. 563–590.

Kastner, R., Gong, W., Hao, X., Brewer, F., Kaplan, A., Brisk, P. and Sarrafzadeh, M. (2006). Layout driven data communication optimization for high level synthesis, in Proceedings of the conference on Design, automation and test in Europe: Proceedings, DATE '06, pp. 1185–1190, European Design and Automation Association, 3001 Leuven, Belgium, Belgium, ISBN 3-9810801-0-6.

Kelly, W., Rosser, E., Pugh, B., Wonnacott, D., Shpeisman, T. and Maslov, V. (2008). The Omega Calculator and Library, version 2.1, available at `http://www.cs.umd.edu/projects/omega/`.

Keutzer, K., Malik, S., Newton, A. R., Rabaey, J. M. and Sangiovanni-Vincentelli, A. (2000). System-Level Design: Orthogonalization of Concerns and Platform-Based Design, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 19, no. 12, pp. 1523–43.

Kienhuis, B., Rijpkema, E. and Deprettere, E. F. (2000). Compaan: Deriving Process Networks from Matlab for Embedded Signal Processing Architectures, in CODES'00, pp. 13–17.

Kim, T. and Liu, X. (2010). A functional unit and register binding algorithm for interconnect reduction, Trans. Comp.-Aided Des. Integ. Cir. Sys., vol. 29, pp. 641–646.

Kim, Y., Kopuri, S. and Mansouri, N. (2004). Automated Formal Verification of Scheduling Process Using Finite State Machines with Datapath (FSMD), in Proceedings of the 5th International Symposium on Quality Electronic Design, ISQED '04, pp. 110–115, IEEE Computer Society, Washington, DC, USA, ISBN 0-7695-2093-6.

Kim, Y. and Mansouri, N. (2008). Automated formal verification of scheduling with speculative code motions, in Proceedings of the 18th ACM Great Lakes symposium on VLSI, GLSVLSI '08, pp. 95–100, ACM, New York, NY, USA, ISBN 978-1-59593-999-9.

King, J. C. (1980). Program correctness: On inductive assertion methods, IEEE Trans. on Software Engineering, vol. SE-6, no. 5, pp. 465–479.

Knoop, J., Ruthing, O. and Steffen, B. (1992). Lazy Code Motion, in PLDI, pp. 224–234.

Krol, T., van Meerbergen, J., Niessen, C., Smits, W. and Huisken, J. (1992). The Sprite Input Language-an intermediate format for high levelsynthesis, in Proceedings. [3rd] European Conference on Design Automation, pp. 186–192.

Kundu, S., Lerner, S. and Gupta, R. (2008). Validating High-Level Synthesis, in Proceedings of the 20th international conference on Computer Aided Verification, CAV '08, pp. 459–472, Springer-Verlag, Berlin, Heidelberg, ISBN 978-3-540-70543-7.

Kundu, S., Lerner, S. and Gupta, R. (2010). Translation Validation of High-Level Synthesis, IEEE Transactions on CAD of ICS, vol. 29, no. 4, pp. 566–579.

Kundu, S., Tatlock, Z. and Lerner, S. (2009). Proving optimizations correct using parameterized program equivalence, in Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation, PLDI ’09, pp. 327–337, ACM, New York, NY, USA, ISBN 978-1-60558-392-1.

Lakshminarayana, G., Khouri, K. and Jha, N. (1997). Wavesched: A novel scheduling technique for control-flow intensive behavioural descriptions, in Proc. of ICCAD, pp. 244–250.

Lakshminarayana, G., Raghunathan, A. and Jha, N. (2000). Incorporating Speculative Execution into Scheduling of Control-Flow-Intensive Design, IEEE Transactions on CAD of ICS, vol. 19, no. 3, pp. 308–324.

Lakshminarayana, G., Raghunathan, A., Jha, N. K. and Dey, S. (1999). Power management in high-level synthesis, IEEE Trans. Very Large Scale Integr. Syst., vol. 7, pp. 7–15.

Lam, M. S. and Wilson, R. P. (1992). Limits of control flow on parallelism, in Proceedings of the 19th annual international symposium on Computer architecture, ISCA ’92, pp. 46–57, ACM, New York, NY, USA, ISBN 0-89791-509-7.

Landwehr, B. and Marwedel, P. (1997). A New Optimization Technique for Improving Resource Exploitation and Critical Path Minimization, in ISSS, pp. 65–72, ACM, New York, NY, USA.

Lee, J.-H., Hsu, Y.-C. and Lin, Y.-L. (1989a). A new integer linear programming formulation for the scheduling problem in data path synthesis, in Procs. of the International Conference on Computer-Aided Design, pp. 20 –23, IEEE Computer Society, Washington, DC, USA.

Lee, J.-H., Hsu, Y.-C. and Y-L, L. (1989b). A New Integer Linear Formulation for the Scheduling Program in High Level Synthesis, in Procs. of the IEEE Conference on Computer-Aided Design, pp. 20–23.

Li, F. and Kandemir, M. (2005). Locality-conscious workload assignment for array-based computations in MPSOC architectures, in Proceedings of DAC ’05, pp. 95–100, ISBN 1-59593-058-2.

Li, P., Agrawal, K., Buhler, J. and Chamberlain, R. D. (2010). Deadlock avoidance for streaming computations with filtering, in Proceedings of the 22nd ACM symposium on Parallelism in algorithms and architectures, SPAA ’10, pp. 243–252, ISBN 978-1-4503-0079-7.

Lin, Z. and Jha, N. (2005). Interconnect-aware low-power high-level synthesis, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 24, no. 3, pp. 336 – 351.

lp_solve (2010).
http://lpsolve.sourceforge.net/5.5/

Majumdar, R. and Xu, R.-G. (2007). Directed test generation using symbolic grammars, in The 6th Joint Meeting on European software engineering conference and the ACM SIGSOFT symposium on the foundations of software engineering: companion papers, ESEC-FSE companion ’07, pp. 553–556, ACM, New York, NY, USA, ISBN 978-1-59593-812-1.

Mandal, A., Mandal, C. and Reade, C. (2007). A System for Automatic Evaluation of C Programs: Features and Interfaces, International Journal of Web-Based Learning and Teaching Technologies, vol. 2, no. 4, pp. 24–39.

Mandal, C. and Chakrabarti, P. P. (2003). Genetic Algorithms for High-Level Synthesis in VLSI Design, Materials and Manufacturing Processes, vol. 18, no. 3, pp. 355–383.

Mandal, C. and Zimmer, R. M. (2000). A Genetic Algorithm for the Synthesis of Structured Data Paths, in 13th International Conference on VLSI Design, pp. 206–211, IEEE Computer Society Press, ISBN 0-7695-0487-6.

Manna, Z. (1974). Mathematical Theory of Computation, McGraw-Hill Kogakusha, Tokyo.

Mansouri, N. and Vemuri, R. (1998). A Methodology for Automated Verification of Synthesized RTL Designs and Its Integration with a High-Level Synthesis Tool, in Proceedings of the Second International Conference on Formal Methods in Computer-Aided Design, FMCAD ’98, pp. 204–221, Springer-Verlag, London, UK, ISBN 3-540-65191-8.

Mansouri, N. and Vemuri, R. (1999). Accounting for various register allocation schemes during post-synthesis verification of RTL designs, in Proceedings of the conference on Design, automation and test in Europe, DATE ’99, pp. 223–230, ACM, New York, NY, USA, ISBN 1-58113-121-6.

Marwedel, P. (2006). Embedded System Design, Springer(India) Private Limited, New Delhi, India.

Meijer, S., Kienhuis, B., Turjan, A. and de Kock, E. (2007). Interactive presentation: A process splitting transformation for Kahn process networks, in DATE ’07: Proceedings of the conference on Design, automation and test in Europe, pp. 1355–1360, ISBN 978-3-9810801-2-4.

Meijer, S., Nikolov, H. and Stefanov, T. (2009). On Compile-time Evaluation of Process Partitioning Transformations for Kahn Process Networks, in Proc. of IEEE/ACM/IFIP Int. Conf. on HW/SW Codesign and System Synthesis (CODES-ISSS’09), pp. 31–40.

Meijer, S., Nikolov, H. and Stefanov, T. (2010a). Combining process splitting and merging transformations for Polyhedral Process Networks, in 8th IEEE Workshop on Embedded Systems for Real-Time Multimedia (ESTIMedia), 2010, pp. 97 –106.

Meijer, S., Nikolov, H. and Stefanov, T. (2010b). Throughput Modeling to Evaluate Process Merging Transformations in Polyhedral Process Networks, in Proc. of Int. Conf. Design, Automation and Test in Europe (DATE’10), pp. 747–752.

Menon, V., Pingali, K. and Mateev, N. (2003). Fractal symbolic analysis, ACM Trans. Program. Lang. Syst., vol. 25, no. 6, pp. 776–813.

Moon, S.-M. and Ebcioğlu, K. (1992). An efficient resource-constrained global scheduling technique for superscalar and VLIW processors, in Proceedings of the 25th annual international symposium on Microarchitecture, MICRO 25, pp. 55–71, IEEE Computer Society Press, Los Alamitos, CA, USA, ISBN 0-8186-3175-9.

Muchnick, S. S. (1997). Advanced compiler design and implementation, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, ISBN 1-55860-320-4.

Musoll, E. and Cortadella, J. (1995). High-level synthesis techniques for reducing the activity of functional units, in Proceedings of the ISLPED ’95, pp. 99–104, ISBN 0-89791-744-8.

Nicolau, A. and Novack, S. (1993). Trailblazing: A Hierarchical Approach to Percolation Scheduling, in ICPP 1993. International Conference on Parallel Processing, 1993, vol. 2, pp. 120 –124.

Nikolov, H., Stefanov, T. and Deprettere, E. (2008). Systematic and Automated Multiprocessor System Design, Programming, and Implementation, IEEE Transactions on CAD of Integrated Circuits and Systems, vol. 27, no. 3, pp. 542–555.

Olson, A. and Evans, B. (2005). Deadlock detection for distributed process networks, in IEEE International Conference on Acoustics, Speech, and Signal Processing, 200, vol. 5, pp. v/73 – v/76 Vol. 5.

Palkovic, M., Catthoor, F. and Corporaal, H. (2009). Trade-offs in loop transformations, ACM Trans. Des. Autom. Electron. Syst., vol. 14, pp. 22:1–22:30.

Panda, P. and Dutt, N. (1995). 1995 high level synthesis design repository, in Proceedings of the 8th international symposium on System synthesis, ISSS ’95, pp. 170–174, ACM, New York, NY, USA, ISBN 0-89791-771-5.

Panda, P. R., Catthoor, F., Dutt, N. D., Danckaert, K., Brockmeyer, E., Kulkarni, C., Vandercappelle, A. and Kjeldsberg, P. G. (2001). Data and memory optimization techniques for embedded systems, ACM Trans. Des. Autom. Electron. Syst., vol. 6, pp. 149–206.

Parker, A. C., Pizarro, J. T. and Mlinar, M. (1986). MAHA: a program for datapath synthesis, in Proceedings of the 23rd ACM/IEEE Design Automation Conference, DAC ’86, pp. 461–466, IEEE Press, Piscataway, NJ, USA, ISBN 0-8186-0702-5.

Parks, T. M. (1995). Bounded Scheduling of Process Networks, Ph.D. thesis, EECS Department, University of California, Berkeley.
http://www.eecs.berkeley.edu/Pubs/TechRpts/1995/2926.html

Paulin, P. G. and Knight, J. P. (1987). Force–Directed Scheduling in Automatic Data Path Synthesis, Procs. of the 24th Design Automation Conference.

Paulin, P. G. and Knight, J. P. (1989). Scheduling and binding algorithms for high-level synthesis, in Proceedings of the 26th ACM/IEEE Design Automation Conference, DAC ’89, pp. 1–6, ACM, New York, NY, USA, ISBN 0-89791-310-8.

Potkonjak, M., Dey, S., Iqbal, Z. and Parker, A. (1993). High performance embedded system optimization using algebraic and generalized retiming techniques, in Proc. of ICCD, pp. 498 –504, IEEE Computer Society, Washington, DC, USA.

Qi, D., Roychoudhury, A., Liang, Z. and Vaswani, K. (2009). Darwin: an approach for debugging evolving programs, in Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering, ESEC/FSE '09, pp. 33–42, ACM, New York, NY, USA, ISBN 978-1-60558-001-2.

Qiu, M., Sha, E. H. M., Liu, M., Lin, M., Hua, S. and Yang, L. T. (2008). Energy minimization with loop fusion and multi-functional-unit scheduling for multidimensional DSP, J. Parallel Distrib. Comput., vol. 68, no. 4, pp. 443–455.

Radhakrishnan, R., Teica, E. and Vermuri, R. (2000). An approach to high-level synthesis system validation using formally verified transformations, in Proceedings of the IEEE International High-Level Validation and Test Workshop (HLDVT'00), HLDVT '00, pp. 80–85, IEEE Computer Society, Washington, DC, USA, ISBN 0-7695-0786-7.

Raghavan, V. (2010). Principles of Compiler Design, Tata McGraw Hill Education Private Limited, New Delhi.

Raghunathan, A., Dey, S. and Jha, N. (1999). Register transfer level power optimization with emphasis on glitch analysis and reduction, IEEE Transactions on CAD of ICS, vol. 18, no. 8, pp. 1114 –1131.

Rajan, S. P. (1995). Correctness of Transformations in High Level Synthesis, in CHDL '95: 12th Conference on Computer Hardware Description Languages and their Applications, pp. 597–603, Chiba, Japan.

Raudvere, T., Sander, I. and Jantsch, A. (2008). Application and Verification of Local Nonsemantic-Preserving Transformations in System Design, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 27, no. 6, pp. 1091 –1103.

Rim, M., Fann, Y. and Jain, R. (1995). Global scheduling with code motions for high-level synthesis applications, IEEE Transactions on VLSI Systems, vol. 3, no. 3, pp. 379–392.

Ruthing, O., Knoop, J. and Steffen, B. (2000). Sparse Code Motion, in IEEE POPL, pp. 170–183.

Sampath, P., Rajeev, A. C., Ramesh, S. and Shashidhar, K. C. (2008). Behaviour Directed Testing of Auto-code Generators, in Proceedings of the 2008 Sixth IEEE International Conference on Software Engineering and Formal Methods, pp. 191–200, IEEE Computer Society, Washington, DC, USA, ISBN 978-0-7695-3437-4.

Samsom, H., Franssen, F., Catthoor, F. and De Man, H. (1995). System level verification of video and image processing specifications, in Proceedings of the 8th international symposium on System synthesis, ISSS ’95, pp. 144–149, ACM, New York, NY, USA, ISBN 0-89791-771-5.

Sarkar, D. and De Sarkar, S. (1989). Some inference rules for integer arithmetic for verification of flowchart programs on integers, IEEE Trans Software. Engg., vol. 15, no. 1, pp. 1–9.

Shashidhar, K. C. (2008). Efficient Automatic Verification of Loop and Data-flow Transformations by Functional Equivalence Checking, Ph.D. thesis, Department of Computer Science, Katholieke Universiteit Leuven, Belgium.

Shashidhar, K. C., Bruynooghe, M., Catthoor, F. and Janssens, G. (2002). Geometric Model Checking: An Automatic Verification Technique for Loop and Data Reuse Transformations, Electronic Notes in Theoretical Computer Science (ENTCS), vol. 65, no. 2, pp. 71–86.

Shashidhar, K. C., Bruynooghe, M., Catthoor, F. and Janssens, G. (2005a). Functional Equivalence Checking for Verification of Algebraic Transformations on Array-Intensive Source Code, in Proc. of DATE’05, pp. 1310–1315, ISBN 0-7695-2288-2.

Shashidhar, K. C., Bruynooghe, M., Catthoor, F. and Janssens, G. (2005b). Verification of Source Code Transformations by Program Equivalence Checking, in 14th International Conference on Compiler Construction (CC’05), pp. 221–236.

Sllame, A. and Drabek, V. (2002). An efficient list-based scheduling algorithm for high-level synthesis, in Euromicro Symposium on Digital System Design, 2002, pp. 316 – 323.

Strehl, K. and Thiele, L. (2000). Interval diagrams for efficient symbolic verification of process networks, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 19, no. 8, pp. 939 –956.

Synopsys (2011). Synphony C Compiler. www.synopsys.com/Systems/BlockDesign/HLS/Pages/SynphonyC-Compiler.aspx

Trickey, H. (1987). Flamel: A High Level Hardware Compiler, IEEE Trans. on CAD., vol. 6, pp. 259–269.

Tsai, F.-S. and Hsu, Y.-C. (1992). STAR - An Automatic Data Path Allocator, IEEE Trans. on CAD., vol. 11, no. 9, pp. 1053–1064.

Tseng, C. J. and Siewiorek, D. P. (1986). Automated Synthesis of Data Paths in Digital-Systems, IEEE Trans. on CAD., vol. 5, no. 3, pp. 379–395.

Turjan, A. (2007). Compiling Nested Loop Programs to Process Networks, Ph.D. thesis, Leiden University.

Turjan, A., Kienhuis, B. and Deprettere, E. (2004). Translating affine nested-loop programs to process networks, in CASES ’04: Proceedings of the 2004 international conference on Compilers, architecture, and synthesis for embedded systems, pp. 220–229, ISBN 1-58113-890-3.

Verdoolaege, S., Janssens, G. and Bruynooghe, M. (2009). Equivalence Checking of Static Affine Programs Using Widening to Handle Recurrences, in Proceedings of CAV ’09, pp. 599–613, ISBN 978-3-642-02657-7.

Verdoolaege, S., Palkovič, M., Bruynooghe, M., Janssens, G. and Catthoor, F. (2010). Experience with Widening Based Equivalence Checking in Realistic Multimedia Systems, J. Electron. Test., vol. 26, no. 2, pp. 279–292.

Viswanath, V., Vasudevan, S. and Abraham, J. (2009). Dedicated Rewriting: Automatic Verification of Low Power Transformations in RTL, in 22nd International Conference on VLSI Design, 2009, pp. 77 –82.

Šimunić, T., Benini, L., De Micheli, G. and Hans, M. (2000). Source code optimization and profiling of energy consumption in embedded systems, in Proceedings of the 13th international symposium on System synthesis, ISSS ’00, pp. 193–198, IEEE Computer Society, Washington, DC, USA, ISBN 1-58113-267-0.

Wakabayashi, K. and Yoshimura, T. (1989). A resource sharing and control synthesis method for conditional branches, in IEEE International Conference on Computer-Aided Design, 1989 (ICCAD-89), pp. 62 –65, IEEE Computer Society, Washington, DC, USA.

Wolf, M. E. and Lam, M. S. (1991). A Loop Transformation Theory and an Algorithm to Maximize Parallelism, IEEE Trans. Parallel Distrib. Syst., vol. 2, no. 4, pp. 452–471.

Xing, X. and Jong, C. C. (2007). A look-ahead synthesis technique with backtracking for switching activity reduction in low power high-level synthesis, Microelectron. J., vol. 38, pp. 595–605.

Xue, L., Ozturk, O. and Kandemir, M. (2007). A memory-conscious code parallelization scheme, in Proceedings of the 44th annual Design Automation Conference, DAC '07, pp. 230–233, ACM, New York, NY, USA, ISBN 978-1-59593-627-1.

Yices (2010).
http://yices.csl.sri.com/

ZamanZadeh, S., Najibi, M. and Pedram, H. (2009). Pre-synthesis Optimization for Asynchronous Circuits Using Compiler Techniques, in Advances in Computer Science and Engineering, vol. 6 of *Communications in Computer and Information Science*, pp. 951–954, Springer Berlin Heidelberg, ISBN 978-3-540-89985-3.

Zhang, C. and Kurdahi, F. (2007). Reducing off-chip memory access via stream-conscious tiling on multimedia applications, Int. J. Parallel Program., vol. 35, no. 1, pp. 63–98.

Zhu, H. W. and Jong, C. C. (2002). Interconnection optimization in data path allocation using minimal cost maximal flow algorithm, Microelectronics Journal, vol. 33, no. 9, pp. 749 – 759.

Zhu, Y., Magklis, G., Scott, M. L., Ding, C. and Albonesi, D. H. (2004). The Energy Impact of Aggressive Loop Fusion, in PACT '04: Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques, pp. 153–164, ISBN 0-7695-2229-7.

Zory, J. and Coelho, F. (1998). Using Algebraic Transformations to Optimize Expression Evaluation in Scientific Code, in Proceedings of the 1998 International

Conference on Parallel Architectures and Compilation Techniques, PACT ’98, pp. 376–384, IEEE Computer Society, Washington, DC, USA, ISBN 0-8186-8591-3.

Zuck, L., Pnueli, A., Goldberg, B., Barrett, C., Fang, Y. and Hu, Y. (2005). Translation and Run-Time Validation of Loop Transformations, Form. Methods Syst. Des., vol. 27, no. 3, pp. 335–360.

Zuck, L., Pnueli, A., Y. Fang and Goldberg, B. (2003). VOC: A translation validator for optimizing compilers, Journal of Universal Computer Science, vol. 9, no. 3, pp. 223–247.