# TRANSLATION VALIDATION OF OPTIMIZING TRANSFORMATIONS OF PROGRAMS USING EQUIVALENCE CHECKING

**Kunal Banerjee**

# TRANSLATION VALIDATION OF OPTIMIZING TRANSFORMATIONS OF PROGRAMS USING EQUIVALENCE CHECKING

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

of

**Doctor of Philosophy**

by

**Kunal Banerjee**

*Under the supervision of*

**Dr. Chittaranjan Mandal**
and
**Dr. Dipankar Sarkar**

**Department of Computer Science and Engineering**

**Indian Institute of Technology Kharagpur**

**July 2015**

## APPROVAL OF THE VIVA-VOCE BOARD

Certified that the thesis entitled **"Translation Validation of Optimizing Transformations of Programs using Equivalence Checking,"** submitted by **Kunal Banerjee** to the Indian Institute of Technology Kharagpur, for the award of the degree of Doctor of Philosophy has been accepted by the external examiners and that the student has successfully defended the thesis in the viva-voce examination held today.

| | |
|---|---|
| Prof Sujoy Ghose<br>(Member of the DSC) | Prof Pallab Dasgupta<br>(Member of the DSC) |
| Prof Santanu Chattopadhyay<br>(Member of the DSC) | |
| Prof Chittaranjan Mandal<br>(Supervisor) | Prof Dipankar Sarkar<br>(Supervisor) |
| (External Examiner) | (Chairman) |

Date:

## CERTIFICATE

This is to certify that the thesis entitled **"Translation Validation of Optimizing Transformations of Programs using Equivalence Checking,"**, submitted by **Kunal Banerjee** 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 this thesis is original and has been done by myself under the general supervision of my supervisors.

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

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

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

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

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

Kunal Banerjee

# ACKNOWLEDGMENTS

Upon embarking on the journey for PhD, I realized that it requires patience, devotion, determination and diligence on a level which I had not experienced before. The fact that I am presently at the stage of submitting my thesis is owed to the contributions of many people. The primary sources of my inspiration have been my supervisors, Dr Chittaranjan Mandal and Dr Dipankar Sarkar. They have always encouraged me to pursue my goals and have extended their help whenever needed. Secondly, I shall like to express my gratitude to my parents who have always supported me. Thirdly, I shall like to thank my elder brother, the first PhD in our family, who has been the primary motivator for me to follow the doctoral degree.

Next, I thank my colleagues in our research group namely, Chandan Karfa, Gargi Roy, K K Sharma, Partha De, Soumyadip Bandyopadhyay, Soumyajit Dey and Sudakshina Dutta, for all the enriching discussions; I consider myself fortunate to have found such colleagues. I also had the opportunity to spend some delightful time with my friends Antara, Arindam, Aritra, Ayan, Bijit, Joy, Krishnendu, Mainak, Rajorshee, Satya Gautam, Sayan, Subhomoy, Subhrangshu, Surajit, and many more; I thank them for their pleasant company. I am grateful to Bappa who has been the key person in sorting out all the office related work. I am also indebted to all the anonymous reviewers of our papers; irrespective of whether the verdict was favourable or not, I always found their criticism helpful and they have contributed much in shaping my thesis. Lastly, I thank Tata Consultancy Services (TCS) for awarding me TCS PhD Fellowship which included contingency grants and travel grants for visiting conferences both within India and abroad; this thesis is largely owed to TCS's generous support.

***Kunal Banerjee***

# ABSTRACT

A compiler translates a source code into a target code, often with an objective to reduce the execution time and/or save critical resources. Thus, it relieves the programmer of the effort to write an efficient code and instead, allows focusing only on the functionality and the correctness of the program being developed. However, an error in the design or in the implementation of the compiler may result in software bugs in the target code generated by that compiler. Translation validation is a formal verification approach for compilers whereby, each individual translation is followed by a validation phase which verifies that the target code produced correctly implements the source code. This thesis presents some translation validation techniques for verifying code motion transformations (while underlining the special treatment required on course to handle the idiosyncrasies of array-intensive programs), loop transformations and arithmetic transformations in the presence of recurrences; it additionally relates two competing translation validation techniques namely, bisimulation relation based approach and path based approach.

A symbolic value propagation based equivalence checking technique over the Finite State Machine with Datapath (FSMD) model has been developed to check the validity of code motion transformations; this method is capable of verifying code motions across loops as well which the previously reported path based verification techniques could not.

Bisimulation relation based approach and path based approach provide two alternatives for translation validation; while the former is beneficial for verifying advanced code transformations, such as loop shifting, the latter surpasses in being able to handle non-structure preserving transformations and guaranteeing termination. We have developed methods to derive bisimulation relations from the outputs of the path based equivalence checkers to relate these competing translation validation techniques.

The FSMD model has been extended to handle code motion transformations of array-intensive programs with the array references represented using McCarthy's read and write functions. This improvement has necessitated addition of some grammar rules in the normal form used for representing arithmetic expressions that occur in the datapath; the symbolic value propagation based equivalence checking scheme is also adapted to work with the extended model.

Compiler optimization of array-intensive programs involves extensive application of loop transformations and arithmetic transformations. A major obstacle for translation validation of such programs is posed by recurrences, which essentially lead to cycles in the data-dependence graphs of the programs making dependence analyses and simplifications of the data transformations difficult. A validation scheme is developed for such programs by isolating the cycles in the data-dependence graphs from the acyclic portions and treating them separately. Thus, this work provides a unified equivalence checking framework to handle loop and arithmetic transformations along with recurrences.

**Keywords:** Translation Validation, Equivalence Checking, Bisimulation Relation, Code Motion Transformation, Loop Transformation, Arithmetic Transformation, Recurrence, Finite State Machine with Datapath (FSMD), Array Data Dependence Graph (ADDG)

# Contents

# List of Symbols

| | | |
|---|---|---|
| $M_1, M_2$ | Finite State Machines with Datapath (FSMDs) | 31 |
| $Q_i$ | Set of control states of FSMD $M_i$ | 30 |
| $q_{i,0}$ | Reset state of FSMD $M_i$ | 30 |
| $I$ | Set of input variables | 30 |
| $V_i$ | Set of storage variables of FSMD $M_i$ | 30 |
| $O$ | Set of output variables | 30 |
| $\tau$ | State transition function of FSMD | 30 |
| $\mathcal{S}$ | Set of status expressions | 30 |
| $\mu$ | Computation in an FSMD | 30 |
| $\mu_1 \simeq \mu_2$ | Computations $\mu_1$ and $\mu_2$ are equivalent | 31 |
| $M_1 \sqsubseteq M_2$ | An FSMD $M_1$ is contained in an FSMD $M_2$ | 31 |
| $q_{1,i}$ | A state of FSMD $M_1$ | 32 |
| $q_{2,i}$ | A state of FSMD $M_2$ | 32 |
| $\alpha, \beta$ | Paths in FSMD | 33 |
| $\alpha^s$ | Start state of path $alpha$ | 31 |
| $\alpha^f$ | End (final) state of path $\alpha$ | 31 |
| $R_\alpha$ | Condition of execution of the path $\alpha$ | 31 |
| $r_\alpha$ | Data transformation of the path $\alpha$ | 31 |
| $\mathsf{s}_\alpha$ | Storage variable transformations of the path $\alpha$ | 31 |
| $\mathsf{s}_\alpha\|_x$ | Storage variable transformations restricted over the variable $x$ | 39 |
| $\theta_\alpha$ | Output list of the path $\alpha$ | 31 |
| $P_1, P_2$ | Path cover of the FSMDs $M_1$ and $M_2$ | 32 |
| $\alpha \simeq \beta$ | Paths $\alpha$ and $\beta$ are unconditionally equivalent | 38 |
| $\alpha \simeq_c \beta$ | Paths $\alpha$ and $\beta$ are conditionally equivalent | 38 |
| $\overline{\vartheta}$ | Propagated vector | 37 |
| $\overline{\vartheta_{i,j}}$ | Propagated vector at state $q_{i,j}$ | 37 |
| $\overline{\upsilon}$ | Initial propagated vector | 37 |
| $\varpi_{i,j}$ | Null path at state $q_{i,j}$ | 44 |
| $\delta$ | Set of corresponding state pairs | 49 |
| $\delta_c$ | Set of conditionally corresponding state pairs | 75 |
| $q_{i,j} \rightarrow q_{i,k}$ | The edge between the states $q_{i,j}$ and $q_{i,k}$ | 52 |
| $q_{i,j} \twoheadrightarrow q_{i,k}$ | The path between the states $q_{i,j}$ and $q_{i,k}$ | 33 |
| $q_{i,j} \overset{c}{\twoheadrightarrow} q_{i,k}$ | The path between the states $q_{i,j}$ and $q_{i,k}$ having $c$ as its condition of execution | 33 |
| $\overline{v}$ | Vector of variables | 68 |

# List of Figures

# List of Tables

# Chapter 1

# Introduction

A *compiler* is a computer program which translates a source code into a target code, often with an objective to reduce the execution time and/or save critical resources. Thus, it relieves the programmer of the duty to write an efficient code and instead, allows one to focus only on the functionality of a program. Hence, designers advocate representation of an initial meta-model of a system in the form of a high-level program which is then transformed through a sequence of optimizations applied by compilers eventually leading to the final implementation. However, an error in the design or in the implementation of a compiler may result in software bugs in the target code obtained from that compiler.

Formal verification can be used to provide guarantees of compiler correctness. It is an attractive alternative to traditional methods of testing and simulation, which tend to be time consuming and suffer from incompleteness. 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 the optimized programs are run even once. Most of the techniques that provide guarantees of correctness by construction require user interaction [105]. Moreover, proofs of correctness by construction are harder to achieve because they must show that *any* application of the optimization is correct. A correct by construction compiler for the C language is targeted by the CompCert compiler [112]. Since it is very hard to formally verify all passes of a more general C compiler such as GCC [4], the optimization passes implemented in CompCert are limited in number. Moreover, undecidability of the general problem of program verification restricts the scope of the input language supported by the verified compiler; for example, the input language supported by CompCert is Clight [29], a subset of the C language. Therefore, such correct by construction compilers are still more of an academic interest and are yet to transcend into industrial practices. It is important to note that 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 behaviour as the original behaviour. The second category of formal verification approach, called *translation validation*, was proposed by Pnueli et al. in [130]; the method consists in proving correctness each time a sequence of optimization steps 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 ensures that any errors in specific instances of translation by the optimizer may be detected, thereby preventing such errors from propagating any further in the synthesis process. Another advantage of the translation validation approach is its scope of validating human expert guided transformations. The present work is aimed at developing translation validation methodologies for several behavioural transformations. Specifically, we focus on proving functional equivalence between the source behaviour and the transformed behaviour. It is to be noted that preservation of non-functional properties, such as timing performance [156], has not been addressed in the current work.

The chapter is organized as follows. Section 1.1 presents a survey of the related literature and brings out the motivation of the work presented in the thesis. Section 1.2 presents an overview of the thesis work and summarizes the contributions made. Finally, the outline of the thesis organization is given in Section 1.3.

## 1.1 Literature survey and motivations

As already discussed, a sequence of transformations may be applied to the source behaviour towards obtaining optimal performance in terms of execution time, energy, etc., in course of generating the final implementation. In this section, we briefly describe several such behavioural transformations that are commonly applied by the compilers and the different verification approaches adopted for their validation.

## 1.1.1 Code motion transformations

Code motion based transformations, whereby operations are moved across basic block boundaries, are used widely in the high-level synthesis tools to improve synthesis results [48, 65, 107]. The objectives of code motions are (i) reduction of number of computations performed at run-time and (ii) minimization of lifetimes of the temporary variables to avoid unnecessary register usage. Based on the above objectives, code motions can be classified into three categories namely, *busy, lazy* and *sparse code motions* [101, 138]. Busy code motions (BCMs) advance the code segments as early as possible. Lazy code motions (LCMs) place the code segments as late as possible to facilitate code optimization. LCM is an advanced optimization technique to remove redundant computations. It also involves common subexpression elimination and loop-invariant code motions. In addition, it can also remove partially redundant computations (i.e., computations that are redundant along some execution paths but not for other alternative paths in a program). BCMs may reduce the number of computations performed but it increases the life time of temporary variables. LCMs optimally cover both the goals. However, code size is not taken into account in either of these two approaches as both these methods lead to code replication. Sparse code motions additionally try to avoid code replication. The effects of several global code motion techniques on system performance in terms of energy, power, etc., have been shown in [66, 81–83].

Some recent works [95, 98, 100, 104, 110] target verification of code motion techniques. Specifically, a Finite State Machine with Datapath (FSMD) model based equivalence checking has been reported in [98] that can handle code transformations confined within basic block boundaries. An improvement based on path recomposition is suggested in [100] to verify speculative code motions where the correctness conditions are formulated in higher-order logic and verified using the PVS theorem prover [6]. In [103, 104], a translation validation approach is proposed for high-level synthesis. Their method establishes a bisimulation relation between a set of points in the initial behaviour and those in the scheduled behaviour under the assumption that the control structure of the behaviour does not change during the synthesis process. An equivalence checking method for scheduling verification was proposed in [95]. This method is applicable even when the control structure of the input behaviour has been modified by the scheduler. The work reported in [110] has identified some false-negative cases of the algorithm in [95] and proposed an algorithm to overcome those limitations. The method of [90] additionally handles non-uniform code motions. The methods proposed in [90, 95, 100, 110] are basically path cover based approaches where each behaviour is decomposed into a finite set of finite paths and equivalence of behaviours are established by showing path level equivalence between two behaviours captured as FSMDs. Changes in structures resulting from optimizing transformations are accounted for by extending path segments in a particular behaviour. However, a path cannot be extended across a loop by definition of path cover [56, 118]. Therefore, all these methods fail in the case of code motions across loops, whereupon some code segment before a loop body is placed after the loop body, or vice-versa. The translation validation for LCMs proposed in [150] is capable of validating code motions across loops. However, this method assumes that there exists an injective function from the nodes of the original code to the nodes of the transformed code. Such a mapping may not hold for practical synthesis tools like SPARK [64]; even if it holds, it is hard to obtain such a mapping from the synthesis tool. Another method [80] that addresses verification of LCMs applied by the GCC compiler [4] involves explicit modification in the compiler code to generate traces that describe the applied optimizations as sequences of predefined transformation primitives which are later verified using the PVS theorem prover. For applying such a method to other compilers, one would require similar expert knowledge about the compiler under test and hence it is not readily compliant. It would be desirable to have a method which can verify code motions across loops without taking any information from the synthesis tools.

### 1.1.2 Alternative approaches to verification of code motion transformations: bisimulation vs path based

Constructing bisimulation relations between programs as a means of translation validation has been an active field of study. Translation validation for an optimizing compiler by obtaining simulation relations between programs and their translated versions was first demonstrated by Necula in [126]. The procedure broadly consists of two algorithms – an inference algorithm and a checking algorithm. The inference algorithm collects a set of constraints (representing the simulation relation) in a forward scan of the two programs and then the checking algorithm checks the validity of these constraints. This work is enhanced by Kundu et al. [103, 104] to validate the high-level synthesis process. Unlike Necula's approach, Kundu et al.'s procedure uses a general theorem prover, rather than specialized solvers and simplifiers, and is thus more modular. A major limitation of these methods [103, 104, 126] is that they cannot handle non-structure preserving transformations such as those introduced by path based schedulers [33, 135]. The path based equivalence checkers [90, 91, 95] do not suffer from this drawback.

On the other hand, transformations such as loop shifting [44] that can be handled by bisimulation based methods of [103, 104] by repeated strengthening of the relation over the data states associated with the related control locations of the source and the target codes still elude the path based equivalence checking methods. This process of strengthening the relation iteratively until a fixed point is reached (i.e., the relation becomes strong enough to imply the equivalence), however, may not terminate. On the contrary, a single pass is used to determine the equivalence/non-equivalence for a pair of paths in the path based approaches and hence, the number of paths in a program being finite, these methods are guaranteed to terminate. Thus, we find that both bisimulation and path based approaches have their own merits and demerits, and therefore, both have found application in the field of translation validation of untrusted compilers. However, the conventionality of bisimulation as the approach for equivalence checking raises the natural question of examining whether path based equivalence checking yields a bisimulation relation or not.

### 1.1.3 Loop transformations and arithmetic transformations

Loop transformations are used to increase instruction level parallelism, improve data locality and reduce overheads associated with executing loops of array-intensive applications [13]. On the other hand, arithmetic transformations are used to reduce the number of computations performed and minimize register lifetimes [131]. Loop transformations together with arithmetic transformations are applied extensively in the domain of multimedia and signal processing applications to obtain better performance in terms of energy, area and/or execution time. The work reported in [30], for example, applies loop fusion and loop tiling to several nested loops and parallelizes the resulting code across different processors for multimedia applications. Minimization of the total energy while satisfying the performance requirements for applications with multi-dimensional nested loops is targeted in [79]. Application of arithmetic transformations can improve the performance of computationally intensive applications as suggested in [131, 165]. Often loop transformation and arithmetic transformation techniques are applied dynamically since application of one may create scope of application of several other techniques. In all these cases, it is crucial to ensure that the intended behaviour of the program has not been altered wrongly during transformation.

Verification of loop transformations in array-intensive programs has drawn a significant amount of investigation. A translation validation approach based on transformation specific rules is proposed in [167] for verification of loop interchange, skewing, tiling and reversal transformations. The main drawback of this approach is that it requires information such as the list of transformations applied and their order of application; however, such information need not be readily available from the synthesis tools. A symbolic simulation based approach is proposed in [119] for verification of loop transformations for programs with no recurrence and with affine indices and bounds. However, this method does not handle any arithmetic transformation that may be applied along with loop transformations. Basically, for multiple occurrences of an array in an expression, this method loses track between each occurrence of that array and its corresponding indices in the presence of arithmetic transformations. Another approach for verifying array-intensive programs can be to use off-the-shelf SMT solvers or theorem provers since the equivalence between two programs can be modeled with a formula such that the validity of the formula implies the equivalence [87]. Although SMT solvers and theorem provers can efficiently handle linear arithmetic, they are not equally suitable for handling non-linear arithmetic. Since array-intensive programs often contain non-linear arithmetic, these tools are found to be inadequate for establishing equivalence of such programs [87]. The works reported in [28, 144, 146] consider a restricted class of programs which must have static control-flows, valid schedules, affine indices and bounds and single assignment forms. In [144, 146], the original and the transformed behaviours are modeled as ADDGs and the correctness of the loop transformations is established by showing the equivalence between the two ADDGs. These works are capable of handling a wide variety of loop transformation techniques without taking any information from the synthesis tools. The method proposed in [153, 154] extends the ADDG model to a dependence graph model to handle recurrences along with associative and commutative operations. 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, common sub-expression elimination, constant unfolding, etc., along with loop transformations. The work reported in [86, 88] furnishes an ADDG based method which compares ADDGs at slice-level rather than path-level as performed in [144] and employs a normalization technique [141] for the arithmetic expressions to verify a wide variety of loop transformations and a wide range of arithmetic transformations applied together in array-intensive programs. However, it cannot verify programs involving recurrences because recurrences lead to cycles in the ADDGs which is otherwise a directed acyclic graph (DAG). The presence of cycles makes the existing data-dependence analysis and simplification (through closed-form representations) of the data transformations in ADDGs inapplicable. Therefore, a unified verification framework for verifying loop and arithmetic transformations in the presence of recurrences would be beneficial.

### 1.1.4 Objectives of the work

The objective of this work is to verify, by way of equivalence checking, the correctness of several behavioural transformations that are applied by compilers and also to relate alternative translation validation techniques namely, bisimulation based approach and path based approach. Specifically, the following objectives were identified:

1. Translation validation of code motion transformations
2. Deriving bisimulation relations from path based equivalence checkers
3. Translation validation of code motions in array-intensive programs
4. Translation validation of loop and arithmetic transformations in the presence of recurrences

## 1.2 Contributions of the thesis

In the following, we outline in brief the contributions of this thesis on each of the objectives identified in Section 1.1.4.

### 1.2.1 Translation validation of code motion transformations

Our initial objective was to develop a unified verification approach for code motion techniques, including code motions across loops, and control structure modifications without requiring any information from the transformation engine. This combination of features had not been achieved by any single verification technique earlier. A preliminary version of our work appears in [21] which has later been modified considerably in [22] to handle speculative code motions and dynamic loop scheduling [135]. In addition to uniform and non-uniform code motion techniques, this work aims at verifying code motions across loops by propagating the (symbolic) variable values through all the subsequent path segments if mismatch in the values of some live variables is detected. Repeated propagation of symbolic values is possible until an equivalent path or a final path segment ending in the reset state is reached. In the latter case, any prevailing discrepancy in values indicates that the original and the transformed behaviours are not equivalent; otherwise they are. The variables whose values are propagated beyond a loop must be invariant to that loop for valid code motions across loops. The loop invariance of such values can be ascertained by comparing the propagated values that are obtained while entering the loop and after one traversal of the loop. The method has been implemented and satisfactorily tested on the outputs of a basic block based scheduler [117], a path based scheduler [33] and the high-level synthesis tool SPARK [64] for some benchmark examples.

Along with non-structure preserving transformations that involve path merging/ splitting (as introduced by the schedulers reported in [33, 135]), the uniform code motion techniques that can be verified by our technique include boosting up, boosting down, duplicating up, duplicating down and useful move – a comprehensive study on the classification of these transformations can be found in [136]; the supported non-uniform code motion techniques include speculation, reverse speculation, safe speculation, code motions across loops, etc. To determine the equivalence of a pair of paths, our equivalence checker employs the normalization process described in [141] to represent the conditions of execution and the data transformations of the paths. This normalization technique further aids in verifying the following arithmetic code transformations: associative, commutative, distributive transformations, copy and constant propagation, common subexpression elimination, arithmetic expression simplification, partial evaluation, constant folding/unfolding, redundant computation elimination, etc. It is important to note that the computational complexity of the method presented in [22] has been analyzed and found to be no worse than that for [90], i.e., the symbolic value propagation (SVP) based method of [22] is capable of handling more sophisticated transformations than [90] without incurring any extra overhead of time complexity; in fact, as demonstrated in [22], the implementation of the SVP based equivalence checker has been found to take less execution time in establishing equivalence than those of the path extension based equivalence checkers, [95] and [90].

### 1.2.2 Deriving bisimulation relations from path based equivalence checkers

In [24], we have shown how a bisimulation relation can be derived from an output of the path extension based equivalence checker [90, 91, 95]. This work has subsequently been extended to derive a bisimulation relation from an output of the SVP based equivalence checker [21, 22] as well. It is to be noted that none of the earlier methods that establish equivalence through construction of bisimulation relations has been shown to tackle code motion across loops; our work demonstrates, for the first time, the existence of a bisimulation relation under such a situation.

### 1.2.3 Translation validation of code motion transformations in array-intensive programs

A significant deficiency of the above mentioned equivalence checkers for the FSMD model is their inability to handle an important class of programs, namely those involving arrays. This is so because the underlying FSMD model does not provide formalism to capture array variables in its datapath. The data flow analysis for array-handling programs is notably more complex than those involving only scalars. For example, consider two sequential statements $a[i] \Leftarrow 10$ and $a[j] \Leftarrow 20$, now if $i = j$ then the second statement qualifies as an *overwrite*, otherwise it does not; unavailability of relevant information to resolve such relationships between index variables may result in exponential number of case analyses. Moreover, obtaining the condition of execution and the data transformation of a path by applying simple substitution as outlined by Dijkstra's weakest precondition computation may become more expensive in the presence of arrays; conditional clauses need to be associated depicting equality/inequality of the index expressions of the array references in the predicate as it gets transformed through the array assignment statements in the path.

In [25], we have introduced a new model namely, Finite State Machine with Datapath *having Arrays* (FSMDA), which is an extension of the FSMD model equipped to handle arrays. To alleviate the problem of determining overwrite/non-overwrite, the SVP based method described in [21, 22] is enhanced to propagate the values assumed by index variables[1] in some path to its subsequent paths (in spite of a match); to resolve the problem of computing the path characteristics, the well-known McCarthy's *read* and *write* functions [120] (originally known as *access* and *change*, respectively) have been borrowed to represent assignment and conditional statements involving arrays that easily capture the sequence of transformations carried out on the elements of an array and also allow uniform substitution policy for both scalars and array variables. An improvisation of the normalization process [141] is also suggested in [25] to represent arithmetic expressions involving arrays in normalized forms.

Experimental results are found to be encouraging and attest the effectiveness of the method [25]. It is pertinent to note that the formalism of our model allows operations in non-single assignment form and data-dependent control flow which have posed serious limitations for other methods that have attempted equivalence checking of array-intensive programs [88, 146, 154]. It is also to be noted that our tool detected a bug in the implementation of copy propagation for array variables in the SPARK [64] tool as reported in [25].

[1]Index variables are basically the "scalar" variables which occur in some index expression of some array variable.

### 1.2.4 Translation validation of loop and arithmetic transformations in the presence of recurrences

Our work reported in [20] provides a unified equivalence checking framework based on ADDGs to handle loop and arithmetic transformations along with recurrences – this combination of features has not been achieved by a single verification technique earlier to the best of our knowledge. The validation scheme proposed here isolates the suitable subgraphs (arising from recurrences) in the ADDGs from the acyclic portions and treats them separately; each cyclic subgraph in the original ADDG is compared with its corresponding subgraph in the transformed ADDG in isolation and if all such pairs of subgraphs are found equivalent, then the entire ADDGs (with the subgraphs replaced by equivalent uninterpreted functions of proper arities) are compared using the conventional technique of [88]. Limitations of currently available schemes are thus overcome to handle a broader spectrum of array-intensive programs.

## 1.3 Organization of the thesis

The rest of the thesis has been organized into chapters as follows.

*Chapter 2* provides a detailed literature survey on different code motion transformations, loop transformations and arithmetic transformations in embedded system specifications along with a discussion on some alternative approaches to code motion validation namely, bisimulation based and path based 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* introduces the FSMD model and the path extension based equivalence checking method over this model. The notion of symbolic value propagation (SVP) is explained and then an SVP based equivalence checking method of FSMDs for verifying code motion transformations with a focus on code motions across loops is described. The correctness and the complexity of the verification procedure are formally treated. A clear improvement over the earlier path extension based methods is also demonstrated through the experimental results.

*Chapter 4* covers the definition of simulation and bisimulation relations between FSMD models and elaborates on the methods of deriving these relations from the outputs of the path extension based equivalence checkers and the outputs of the SVP based equivalence checkers.

*Chapter 5* introduces the FSMDA model with the array references and operations represented using McCarthy's read and write functions. The method of obtaining the characteristic tuple of a path in terms of McCarthy's functions is explained next. The enhancements needed in the normalization technique to represent array references is elucidated. The modified SVP based equivalence checking scheme for the FSMDA model is then illustrated. The chapter provides a theoretical analysis of the method. Finally, an experiment involving several benchmark problems has been described.

*Chapter 6* introduces the ADDG model and the related equivalence checking method. Then it covers the extension of the ADDG based equivalence checking technique to handle array-intensive programs which have undergone loop and arithmetic transformations in the presence of recurrences. The correctness of the proposed method is formally proved and the complexity is analyzed subsequently. The superior performance of our method in comparison to other competing methods is also empirically established.

*Chapter 7* concludes our study in the domain of translation validation of optimizing transformations of programs that are applied by compilers and discusses some potential future research directions.

# Chapter 2

# Literature Survey

## 2.1 Introduction

An overview of important research contributions in the area of optimizing transformations is provided in this chapter. For each class of transformations targeted by this thesis, (i) we first study several applications of these transformations to underline their relevance is system design, (ii) we then survey existing verification methodologies for these optimizing transformations and in the process identify their limitations to emphasize on the prominent gaps in earlier literature which this thesis aims to fill. For the alternative approaches to translation validation of code motion transformations namely, bisimulation based method and path based method, we discuss about their various applications in the field of verification.

## 2.2 Code motion transformations

### 2.2.1 Applications of code motion transformations

Designing parallelizing compilers often require application of code motion techniques [10, 54, 61, 74, 101, 108, 124, 127, 138]. Recently, code motion techniques have been successfully applied during scheduling in high-level synthesis. Since the compilers investigated in this thesis are broadly from the domain of embedded system design, in the following, we study the applications of code motion techniques in the context of embedded system design, especially high-level synthesis.

The works reported in [47, 48, 136] support generalized code motions during scheduling in synthesis systems, whereby operations can be moved globally irrespective of their position in the input. These works basically search the solution space and determine the cost associated with each possible solution; eventually, the solution with the least cost is selected. To reduce the search time, the method of [47, 48] proposes a pruning technique to intelligently select the least cost solution from a set of candidate solutions.

Speculative execution is a technique that allows a superscalar processor to keep its functional units as busy as possible by executing instructions before it is known that they will be needed, i.e., some computations are carried out even before the execution of the conditional operations that decide whether they need to be executed at all. The paper [107] presents techniques to integrate speculative execution into scheduling during high-level synthesis. This work shows that the paths for speculation need to be decided according to the criticality of individual operations and the availability of resources in order to obtain maximum benefits. It has been demonstrated to be a promising technique for eliminating performance bottlenecks imposed by control flows of programs, thus resulting in significant gains (up to seven-fold) in execution speed. Their method has been integrated into the Wavesched tool [106].

A global scheduling technique for superscalar and VLIW processors is presented in [123]. This technique targets parallelization of sequential code by removing anti-dependence (i.e., write after read) and output dependence (i.e., write after write) in the data flow graph of a program by renaming registers, as and when required. The code motions are applied globally by maintaining a data flow attribute at the beginning of each basic block which designates what operations are available for moving up through this basic block. A similar objective is pursued in [41]; this work combines the speculative code motion techniques and parallelizing techniques to improve scheduling of control flow intensive behaviours.

In [77], 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 tries to overcome this limitation by combining these two phases of high-level synthesis. This problem is further addressed in [26]; this method analyzes a program to identify the live range overlaps for all possible placements of instructions in basic blocks and all orderings of instructions within blocks and based on this information, the authors formulate an optimization problem to determine code motions and partial local schedules that minimize the overall cost of live range overlaps. The solutions of the formulated optimization problem are evaluated using integer linear programming, where feasible, and a simple greedy heuristic. A method for elimination of parallel copies using code motion on data dependence graphs to optimize register allocation can be found in [31].

The effectiveness of traditional compiler techniques employed in high-level synthesis of synchronous circuits is studied for asynchronous circuit synthesis in [161]. 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.

Benefits of applying code motions to improve results of high-Level synthesis has also been demonstrated in [63, 65, 66], where the authors 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 [65, 69, 70]. They present a scheduling heuristic that guides these code motions and improves scheduling results (in terms of schedule length and finite state machine states) and logic synthesis results (in terms of circuit area and delay) by up to 50 percent. In [62, 63], two novel strategies are presented to increase the scope for application of speculative code motions: (i) 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 and (ii) determining if an operation can be conditionally speculated into multiple basic blocks either by using existing idle resources or by creating new scheduling steps; this strategy leads to balancing of the number of steps in the conditional branches without increasing the longest path through the conditional block. Classical common sub-expression elimination (CSE) technique fails 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. This observation is utilized in [68] and a new approach called *dynamic common sub-expression elimination* is introduced. The code motion techniques and heuristics described in this paragraph have been implemented in the SPARK high-level synthesis framework [64].

Energy management is of concern to both hardware and software designers. An energy-aware code motion framework for a compiler is explained in [159] which basically tries to cluster accesses to input and output buffers, thereby extending the time period during which the input and output buffers are clock or power gated. Another method [114] attempts to change the data access patterns in memory blocks by employing code motions in order to improve the energy efficiency and performance of STT-RAM based hybrid cache. Some insights into how code motion transformations may aid in the design of embedded reconfigurable computing architectures can be found in [46].

### 2.2.2 Verification of code motion transformations

Recently, in [125], a proof construction mechanism has been proposed to verify a few transformations performed by the LLVM compiler [5]; these proofs are later checked for validity using the Z3 theorem prover [9]. Formal verification of single assignment form based optimizations for the LLVM compiler has been addressed in [163]. Similar to Section 2.2.1, henceforth we shall focus on the verification strategies targeting validation of code motions of embedded system specifications.

A formal verification of scheduling process using the FSMD model is reported in [99]. In this paper, cut-points are introduced in both the FSMDs followed by construction of the respective path covers. Subsequently, for every path in one FSMD, an equivalent path in the other FSMD is searched for. Their method requires that the control structure of the input FSMD is not disturbed by the scheduling algorithm and code has not moved beyond basic block boundaries. This implies that the respective path covers obtained from the cut-points are essentially bijective. This requirement, however, imposes a restriction that does not necessarily hold because the scheduler may merge the paths of the original specification into one path of the implementation or distribute operations of a path over various paths for optimization of time steps.

A Petri net based verification method for checking the correctness of algorithmic transformations and scheduling process in high-level synthesis is proposed in [36]. 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 high-level synthesis result is claimed as a correct solution.

All these verification approaches, however, are well suited for basic block based scheduling [76, 111], where the operations are not moved across the basic block boundaries and 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 they entail code being moved from one basic block to other basic blocks.

Some recent works, such as, [100, 103, 104] target verification of code motion techniques. Specifically, a path recomposition based FSMD equivalence checking has been reported in [100] to verify speculative code motions. The correctness conditions are formulated in higher-order logic and verified using the PVS theorem prover [6]. Their path recomposition over conditional blocks fails if non-uniform code motion transformations are applied by the scheduler. In [103, 104], 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 [33]. An equivalence checking method for scheduling verification is given in [95]. 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. In [17], another Petri net based verification strategy is described which represents high-level synthesis benchmarks as untimed *Petri net based Representation of Embedded Systems (PRES+)* models [43] by first translating them into FSMD models and subsequently feeding them to the FSMD equivalence checker of [95]. The work reported in [110] has identified some false-negative cases of the algorithm in [95] and proposed an algorithm to overcome those limitations. This method is further extended in [90] to handle non-uniform code motions as well. An equivalence checking method for ensuring the equivalence between an algorithm description and a behavioural register transfer language (RTL) modeled as FSMDs is given in [71].

None of the above mentioned techniques has been demonstrated to handle code motions across loops. Hence, it would be desirable to have an equivalence checking method that would encompass the ability to verify code motions across loops along with uniform and non-uniform code motions and transformations which alter the control structure of a program.

## 2.3 Alternative approaches to verification of code motion transformations: bisimulation and path based

### 2.3.1 Bisimulation based verification

Transition systems are used to model software and hardware at various abstraction levels. The lower the abstraction level, the more implementation details are present. It is important to verify that the refinement of a given specification retains the intended behaviour. Bisimulation equivalence aims to identify transition systems with the same branching structure, and which thus can simulate each other in a step-wise manner [16]. In essence, a transition system $T$ can simulate transition system $T'$ if every step of $T$ can be matched by one (or more) steps in $T'$. Bisimulation equivalence denotes the possibility of mutual, step-wise simulation. Initially, bisimulation equivalence was introduced as a binary relation between transition systems over the same set of atomic propositions, e.g., bisimulation equivalence between communicating systems as defined by Milner in [122]. However, formulating bisimulation relations in terms of small steps (such as, individual atomic propositions) makes it difficult to reason directly about large program constructs, such as loops and procedures, for which a big step semantics is more natural [102].

Bisimulation based verification has found applications in various fields, such as labeled transition systems [53], concurrent systems [45], timed systems [149], well-structured graphs [49], probabilistic processes [14, 15]. Scalability issues of bisimulation based approaches have been tackled in [37, 55, 157]. A comprehensive study on bisimulation can be found in [16, 140]. Henceforth, we focus on bisimulation based techniques which target verification of code motion transformations.

An automatic verification of scheduling by using symbolic simulation of labeled segments of behavioural descriptions has been proposed in [51]. 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.

In [126], translation validation for an optimizing compiler by obtaining simulation relations between programs and their translated versions was first demonstrated. This procedure broadly consists of two algorithms – an inference algorithm and a checking algorithm. The inference algorithm collects a set of constraints (representing the simulation relation) in a forward scan of the two programs and then the checking algorithm checks the validity of these constraints. Building on this foundation, the authors of [103, 104] have validated the high-level synthesis process. Unlike the method of [126], their procedure takes into account statement-level parallelism since hardware is inherently concurrent and one of the main tasks that high-level synthesis tools perform is exploiting the scope of parallelizing independent operations. Furthermore, the algorithm of [103, 104] uses a general theorem prover, rather than specialized solvers and simplifiers (as used by [126]), and is thus more modular. Advanced code transformations, such as loop shifting [44], can be verified by [103, 104], albeit at the cost of foregoing termination property of their verification algorithm. A major limitation of these methods [103, 104, 126] is that they cannot handle non-structure preserving transformations such as those introduced by path based schedulers [33, 135]; in other words, the control structures of the source and the target programs must be identical if one were to apply these methods. This limitation is alleviated to some extent in [115]. The authors of [115] have studied and identified what kind of modifications the control structures undergo on application of some path based schedulers and based on this knowledge they try to establish which control points in the source and the target programs are to be correlated prior to generating the simulation relations. The ability to handle control structure modifications which are applied by [135], however, still remain beyond the scope of currently known bisimulation based techniques.

### 2.3.2 Path based equivalence checking

Path based equivalence checking was proposed as a means of translation validation for optimizing compilers. Consequently, much of it has already been covered in Section 2.2.2. Nevertheless, we provide a gist of path based equivalence checking strategies for the sake of completeness. However, to avoid repetition, here we only underline its salient features and thereby highlight its complementary advantages as compared to bisimulation based verification. Path based equivalence checking was first proposed in [91], whereby the source and the transformed programs are represented as FSMDs segmented into paths, and for every path in an FSMD, an equivalent path is searched for in the other FSMD; on successful discovery of pairs of equivalent paths such that no path in either FSMD remains unmatched, the two FSMDs are declared equivalent. This method is demonstrated to handle complicated modifications of the control structures introduced by path based scheduler [33] as well as [135]. This method is further enhanced in [90, 95, 110] to increase its power of handling diverse code motion transformations. Its prowess in verifying optimizations applied during various stages of high-level synthesis has been displayed in [89, 92–94, 97]. Note that when a path from a path cover is checked for equivalence, the number of paths yet to be checked for equivalence in that path cover decreases by one; also, the number of paths in a path cover is finite; as a result, all the path based equivalence checking methods are guaranteed to terminate. The loop shifting code transformation [44], however, cannot yet be verified by modern path based equivalence checkers.

Thus, we find that bisimulation based verification and path based equivalence checking have complementary merits and demerits. While the former beats the latter in its ability of handle loop shifting, the latter is more proficient in handling non-structure preserving transformations and, unlike the former, is guaranteed to terminate. Accordingly, both have found applications in the domain of translation validation. However, bisimulation being a more conventional approach, it may be worthwhile to investigate whether an (explicit) bisimulation relation can be derived from the outputs of path based equivalence checkers. On a similar note, a *relation transition system* model is proposed in [73] to combine the benefits of Kripke logical relations and bisimulation relations to reason about programs.

## 2.4 Loop transformations and arithmetic transformations

### 2.4.1 Applications of loop transformations

Loop transformations along with 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.

The effects of loop transformations on system power has been studied extensively. In [82], the impact of loop tiling, loop unrolling, loop fusion, loop fission and scalar expansion on energy consumption has been underlined. In [81], it has been demonstrated that conventional data locality oriented code transformations are insufficient for minimizing disk power consumption. The authors of [81] instead 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 propose a unified optimizer that targets disk power management by applying these transformations. The work reported in [83] exhibits 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. 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. A novel memory-conscious loop parallelization strategy with the objective of minimizing the data memory requirements of processors has been suggested in [158]. The work in [78] presents a data space-oriented tiling (DST) approach. 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. A more global approach to identify data locality problem is taken in [113] which proposes a compiler driven data locality optimization strategy in the context of embedded MP-SoCs. 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 reported in [35] shows 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 conveys how the balance between power saving and reliability improvement can be achieved using a metric called the energy-delay-fallibility product.

Loop transformations have found application in the design of system memory as well. For example, in [30], a technique is proposed 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 [162] 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. It has been known that external memory bandwidth is a crucial bottleneck in the majority of computation-intensive applications for both performance and power consumption. Data reuse is an important technique for reducing the external memory access by utilizing the memory hierarchy. Loop transformation for data locality and memory hierarchy allocation are two major steps in data reuse optimization flow. The paper presented in [39] provides a combined approach which optimizes loop transformation and memory hierarchy allocation simultaneously to achieve global optimal results on external memory bandwidth and on-chip data reuse buffer size. This work is enhanced in [40] for optimizing the on-chip memory allocation by loop transformations in the imperfectly nested loops.

A method to minimize the total energy while satisfying the performance requirements for application with multi-dimensional nested loops was proposed in [85]. 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 [134] is also the same as that of [85]. However, they apply loop fusion and multi-functional unit scheduling techniques to achieve that.

In [60], 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 [129]. The IMEC group [34, 128] pioneered the work on program transformations to reduce energy consumption in data dominated embedded applications. In [57], loop fusion technique is used to optimize multimedia applications before the hardware/software partitioning to reduce the use of temporary arrays. Loop transformations have also been applied to improve performance in coarse-grained reconfigurable architecture [116]. Applications of loop transformations to parallelize sequential code targeting embedded multi-core systems are given in [132, 160]. Several other loop transformation techniques and their effects on embedded system design may be found in [32, 52, 59, 155, 164].

### 2.4.2 Applications of arithmetic transformations

Compiler optimizations often involve 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. Application of retiming, algebraic and redundancy manipulation transformations to improve the performance of embedded systems is proposed in [131].

They introduced a new negative retiming technique to enable algebraic transformations to improve latency/throughput. In [165], 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. They, basically, propose to replace traditional associative commutative pattern-matching techniques which suffer from scalability issues by two performance enhancing algorithms providing factorization and multiply-add extraction heuristics and choice criteria based on a simple cost model. Operation cost minimization by loop-invariant code motion and operator strength reduction is proposed in [67] to achieve minimal code execution within loops and reduced operator strengths. The effectiveness of such source-level transformations is demonstrated in [67] with two real-life multimedia application kernels by comparing the improvements in the number of execution cycles, before and after applying the optimizations. Application of algebraic transformations to minimize critical path length in the domain of computationally intensive applications is proposed in [109]. 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. A set of transformations such as common sub-expression 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 [64, 66]. The potential of arithmetic transformations on FPGAs is studied in [50]. 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 FPGAs whereas expression splitting and resource sharing reduce the clock period of the circuits.

### 2.4.3 Verification of loop and arithmetic transformations

Verification of loop transformations on array-intensive programs is a well studied problem. Some of these target transformation specific verification rules. For example, the methods of [166, 167] proposed permutation rules for verification of loop interchange, skewing, tiling, reversal transformations in their translation validation approach. The rule set is further enhanced in [27, 72]. The main drawback of this approach is that the method had to rely on the hint provided by the compiler. 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 necessitates enhancement of the rule set.

A method called fractal symbolic analysis has been proposed in [121]. 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 [166, 167]. This method combines some of the power of symbolic analysis with the tractability of dependence analysis. The power of this method again depends on the availability of the rule set.

A fully automatic verification method for loop transformations is given in [139]. They have used data dependence analysis and shown the preservation of the dependencies in the original and in the transformed program. The program representation used in this work allows only a statement-level equivalence checking between the programs. Therefore, this method cannot handle arithmetic transformations. 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. Therefore, direct correspondence between the statement classes of the original and the transformed programs does not always hold as required by [139].

The method developed in [144–147] considers a restricted class of programs which must have static control flow, valid schedule, affine indices and bounds, and single assignment form. The authors have proposed an equivalence checking method for verification of loop transformations, where the original and the transformed programs are modeled as Array Data Dependence Graphs (ADDGs). This method is promising since it is capable of handling most of the loop transformation techniques without taking any information from the compilers. The main limitations of the ADDG based method are its inability to handle the following cases: recurrences, data-dependent assignments and accesses, and arithmetic transformations. The method proposed in [153, 154] extends the ADDG model to a dependence graph to handle recurrences and also additionally verifies the associative and the commutative data transformations.

SMT solvers such as CVC4 [3], Yices [7] or theorem provers such as ACL2 [1] can also be used to verify loop transformations. The equivalence between two programs can be modeled with a formula such that the validity of the formula implies the equivalence [87]. Although the SMT solvers and the theorem prover can efficiently handle linear arithmetic, they are not equally potent in handling non-linear arithmetic. Since array-intensive programs (with loops) often contain non-linear arithmetic, these tools are not efficient in handling equivalence of such programs [87].

All the above methods 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 subexpression elimination, etc., along with loop transformations. The definition of equivalence of ADDGs proposed in [144, 146] cannot be extended easily (as in the cases of commutative and associative transformations) to handle these arithmetic transformations. Consequently, a slice-level equivalence of ADDGs is proposed in [86, 88] (as opposed to path-level equivalence of [144, 146]), which additionally incorporates a normalization technique [141] extended suitably to represent data transformations. This method is found capable of establishing equivalence in the presence of both loop and arithmetic transformations. It has also been used in checking correctness of process network level transformations for multimedia and signal processing applications [96]. The initial and the transformed behaviors are both modeled as ADDGs and the verification problem is posed as checking of equivalence between the two ADDGs. This technique, however, cannot handle recurrences because recurrences lead to cycles in the data-dependence graph of a program which make dependence analyses and simplifications (through closed-form representations) of the data transformations difficult.

Hence, it would be desirable to develop a unified equivalence checking framework to handle loop and arithmetic transformations along with recurrences.

## 2.5 Conclusion

In this chapter, we have discussed several applications of code motion transformations, loop transformations and arithmetic transformation, which are applied by contemporary compilers, especially in embedded system design. The state-of-the-art verification methods for these transformations are also discussed in this chapter. In the 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. Two alternative schemes for verifying code motion transformations have also been discussed: bisimulation based verification and path based equivalence checking. Merits and demerits of both the approaches have been underlined. A way to relate these apparently different verification strategies has also been explored in a subsequent chapter.

# Chapter 3

# Translation Validation of Code Motion Transformations

## 3.1 Introduction

The objective of the current work is to develop a unified verification procedure for code motion techniques, including code motions across loop, and control structure modifications without taking any additional information from the compiler. It is important to note that although code optimized using verified compilers render verification at a later stage (such as the one presented in this chapter) unnecessary, such compilers rarely exist and often deal with input languages with considerable restrictions. As a result, these compilers are not yet popular with the electronic design automation industry, thus necessitating behavioural verification as a post-synthesis process. Moreover, often a *verified* compiler is built out of unverified code transformers; in such cases, it is augmented with a proof-assistant to ensure the correctness of the applied transformations in terms of equivalence between the input program and the program obtained by these code transformers, such as [150]; the methodology presented in the current chapter can be used to build such proof-assistants as well.

The chapter is organized as follows. In Section 3.2, the FSMD model and related concepts are described. Section 3.3 illustrates the basic concepts of the symbolic value propagation method with the help of an example. Some intricacies of the method – detection of loop invariance of subexpressions and subsumption of conditions of execution of the paths are also highlighted in this section. The correctness of symbolic value propagation as a method of equivalence checking is given in Section 3.4. The overall verification process is presented in Section 3.5 along with some illustrative examples. Its correctness and complexity are formally treated in Section 3.6. Experimental results are provided in Section 3.7. The chapter is concluded in Section 3.8.

## 3.2 The FSMD model and related concepts

A brief formal description of the FSMD model is given in this section; a detailed description of FSMD models can be found in [95]. The FSMD model [58], used in this work to model the initial behaviour and the transformed behaviour, is formally defined as an ordered tuple $\langle Q, q_0, I, V, O, \tau : Q \times 2^{\mathcal{S}} \to Q,\ h : Q \times 2^{\mathcal{S}} \to U \rangle$, where $Q$ is the finite set of control states, $q_0$ is the reset (initial) state, $I$ is the set of input variables, $V$ is the set of storage variables, $O$ is the set of output variables, $\tau$ is the state transition function, $\mathcal{S}$ represents a set of status signals as relations between two arithmetic expressions over the members of $I$ and $V$, $U$ represents a set of assignments of expressions over inputs and storage variables to some storage or output variables and $h$ is the update function capturing the conditional updates of the output and storage variables taking place in the transitions through the members of $U$.

A *computation* of an FSMD is a finite walk from the reset state back to itself without having any intermediary occurrence of the reset state. The condition of execution $R_\mu$ of a computation $\mu$ is a logical expression over the variables in $I$ such that $R_\mu$ is satisfied by the initial data state iff the computation $\mu$ is executed. The data transformation $r_\mu$ of the computation $\mu$ is the tuple $\langle \mathrm{s}_\mu, \theta_\mu \rangle$; the first member $\mathrm{s}_\mu$ represents the values of the program variables in $V$ in terms of the input variables $I$ at the end of the computation $\mu$ and the second member $\theta_\mu$ represents the output list of the computation $\mu$. To determine the equivalence of arithmetic expressions under associative, commutative, distributive transformations, expression simplification, constant folding, etc., we rely on the normalization technique presented in [141] which supports Booleans and integers only and assumes that no overflow or underflow occurs. The method in [150], in contrast, can handle floating point expressions as well since it treats LCM to be a purely syntactical redundancy elimination transformation.

**Definition 1** (Computation Equivalence). *Two computations $\mu_1$ and $\mu_2$ are said to be* *equivalent, denoted as $\mu_1 \simeq \mu_2$, iff $R_{\mu_1} \equiv R_{\mu_2}$ and $r_{\mu_1} = r_{\mu_2}$.*

**Definition 2** (FSMD Containment)**.** *An FSMD $M_1$ is said to be contained in an FSMD $M_2$, symbolically $M_1 \sqsubseteq M_2$, if for any computation $\mu_1$ of $M_1$ on some inputs, there exists a computation $\mu_2$ of $M_2$ on the same inputs such that $\mu_1 \simeq \mu_2$.*

**Definition 3** (FSMD Equivalence)**.** *Two FSMDs $M_1$ and $M_2$ are said to be computationally equivalent, if $M_1 \sqsubseteq M_2$ and $M_2 \sqsubseteq M_1$.*

However, an FSMD may consist of indefinitely long computations because of presence of loops. So, the idea of directly comparing the computations of the FSMDs exhaustively will not work in practice. Therefore, cut-points are introduced such that each loop is cut in at least one cut-point thereby permitting an FSMD to be considered as a combination of paths.

A *path* $\alpha$ in an FSMD model is a finite sequence of states where the first and the last states are cut-points and there are no intermediary cut-points and any two consecutive states in the sequence are in $\tau$. The initial (start) and the final control states of a path $\alpha$ are denoted as $\alpha^s$ and $\alpha^f$, respectively. The *condition of execution $R_\alpha$ of the path* $\alpha$ is a logical expression over the variables in $V$ and the inputs $I$ such that $R_\alpha$ is satisfied by the (initial) data state of the path iff the path $\alpha$ is traversed. The *data transformation $r_\alpha$ of the path* $\alpha$ is the tuple $\langle s_\alpha, \theta_\alpha \rangle$; the first member $s_\alpha$ 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 $\theta_\alpha$, which represents the output list along the path $\alpha$, is typically of the form $[OUT(P_{i_1}, e_1), OUT(P_{i_2}, e_2), \ldots]$. More specifically, for every expression $e$ output to port $P$ along the path $\alpha$, there is a member $OUT(P, e)$ in the list appearing in the order in which the outputs occur in $\alpha$. The condition of execution and the data transformation of a path are computed using the method of symbolic execution.

**Definition 4** (Path Equivalence)**.** *Two paths $\alpha$ and $\beta$ are said to be computationally equivalent, denoted as $\alpha \simeq \beta$, iff $R_\alpha \equiv R_\beta$ and $r_\alpha = r_\beta$.*

It is worth noting that if two behaviors are to be computationally equivalent, then their outputs must match. So, when some variable is output, its counterpart in the other FSMD must attain the same value. In other words, equivalence of $\theta_\alpha$ hinges upon the equivalence of $s_\alpha$. Hence, the rest of the chapter focuses on computation of $s_\alpha$; the computation of $\theta_\alpha$ has deliberately been omitted for the sake of brevity.

Any computation $\mu$ of an FSMD $M$ can be considered as a computation along some concatenated path $[\alpha_1\alpha_2\alpha_3...\alpha_k]$ of $M$ such that, for $1 \leq i < k$, $\alpha_i$ terminates in the initial state of the path $\alpha_{i+1}$, the path $\alpha_1$ emanates from and the path $\alpha_k$ terminates in the reset state $q_0$ of $M$; $\alpha_i$'s may not all be distinct. Hence, we have the following definition.

**Definition 5** (Path Cover of an FSMD). *A finite set of paths $P = \{\alpha_1, \alpha_2, \alpha_3, \ldots, \alpha_k\}$ is said to be a path cover of an FSMD M if any computation $\mu$ of M can be expressed as a concatenation of paths from P.*

The set of all paths from a cut-point to another cut-point without having any intermediary cut-point is a path cover of the FSMD [56]. In course of establishing equivalence of paths of a path cover, a natural correspondence between the control states from the two FSMDs is also produced as defined below.

**Definition 6** (Corresponding States). *Let $M_1 = \langle Q_1, q_{1,0}, I, V_1, O, \tau_1, h_1\rangle$ and $M_2 = \langle Q_2, q_{2,0}, I, V_2, O, \tau_2, h_2\rangle$ be two FSMDs having identical input and output sets, I and O, respectively.*

1) *The respective reset states $q_{1,0}$ and $q_{2,0}$ are corresponding states and a non-reset state does not have correspondence with a reset state.*

2) *If $q_{1,i} \in Q_1$ and $q_{2,j} \in Q_2$ are corresponding states and there exist $q_{1,k} \in Q_1$ and $q_{2,l} \in Q_2$ such that, for some path $\alpha$ from $q_{1,i}$ to $q_{1,k}$ in $M_1$, there exists a path $\beta$ from $q_{2,j}$ to $q_{2,l}$ in $M_2$ such that $\alpha \simeq \beta$, then $q_{1,k}$ and $q_{2,l}$ are corresponding states (note that $q_{1,k}$ and $q_{2,l}$ must both be either reset states or non-reset states).*

The following theorem can be concluded from the above discussion.

**Theorem 1.** *An FSMD $M_1$ is contained in another FSMD $M_2$ ($M_1 \sqsubseteq M_2$), if there exists a finite path cover $P_1 = \{\alpha_1, \alpha_2, \ldots, \alpha_{l_1}\}$ of $M_1$ for which there exists a set $P_2 = \{\beta_1, \beta_2, \ldots, \beta_{l_2}\}$ of paths of $M_2$ such that for any corresponding state pair $\langle q_{1,i}, q_{2,j}\rangle$, for any path $\alpha_m \in P_1$ emanating from $q_{1,i}$, there exists a path $\beta_n \in P_2$ emanating from $q_{2,j}$ such that $\alpha_m \simeq \beta_n$.*

*Proof:* We say that $M_1 \sqsubseteq M_2$, if for any computation $\mu_1$ of $M_1$ on some inputs, there exists a computation $\mu_2$ of $M_2$ on the same inputs such that $\mu_1 \simeq \mu_2$ (Definition 2). Let there exist a finite path cover $P_1 = \{\alpha_1, \alpha_2, \ldots, \alpha_{l_1}\}$ of $M_1$. Corresponding to $P_1$, let a set $P_2 = \{\beta_1, \beta_2, \ldots, \beta_{l_2}\}$ of paths of $M_2$ exist such that for any corresponding state pair $\langle q_{1,i}, q_{2,j}\rangle$, for any path $\alpha_m \in P_1$ emanating from $q_{1,i}$, there exists a path $\beta_n \in P_2$ emanating from $q_{2,j}$ such that $\alpha_m \simeq \beta_n$.

Since $P_1$ is a path cover of $M_1$, any computation $\mu_1$ of $M_1$ can be looked upon as a concatenated path $[\alpha_{i_1}\alpha_{i_2}\ldots\alpha_{i_t}]$ from $P_1$ starting from the reset state $q_{1,0}$ and ending again at this reset state of $M_1$. Now, we have to show that a computation $\mu_2$ exists in $M_2$ such that $\mu_1 \simeq \mu_2$.

The reset states $q_{1,0}$ of $M_1$ and $q_{2,0}$ of $M_2$ must be corresponding states by clause 1 of Definition 6. Therefore, it follows from the hypothesis that a path $\beta_{j_1}$ exists in $P_2$ such that $\alpha_{i_1} \simeq \beta_{j_1}$; thus, the states $\alpha^f_{i_1}$ and $\beta^f_{j_1}$ must again be corresponding states by clause 2 in Definition 6. By repetitive application of the above argument, it follows that that there exists a concatenated sequence of paths $\beta_{j_1}\ldots\beta_{j_t}$ such that $\alpha_{i_k} \simeq \beta_{j_k}, 1 \leq k \leq t$. What remains to be proved for $[\beta_{j_1}\beta_{j_2}\ldots\beta_{j_t}]$ to be a computation of $M_2$ is that $\beta^f_{j_t} = q_{2,0}$. Let $\beta^f_{j_t} \neq q_{2,0}$; now $\langle \alpha^f_{i_t}, \beta^f_{j_t}\rangle$, i.e., $\langle q_{1,0}, \beta^f_{j_t}\rangle$ must be a corresponding state pair. However, by Definition 6, a non-reset state cannot have correspondence with a reset state. Consequently, $\beta^f_{j_t}$ must be $q_{2,0}$ and thus $[\beta_{j_1}\beta_{j_2}\ldots\beta_{j_t}]$ is a computation, $\mu_2$ say, and $\mu_1 \simeq \mu_2$. ■

It is important to note that the choice of cut-points is non-unique and it is not guaranteed that a path cover of one FSMD obtained from any choice of cut-points in itself will have the corresponding set of equivalent paths for the other FSMD. The following example explains how one's initial choice of cut-points may be revised by the path extension based equivalence checking method [90, 95, 110], which uses Theorem 1 to establish equivalence between two FSMDs.

**Example 1.** Figure 3.1(a) shows the source FSMD $M_1$ and Figure 3.1(b) shows the transformed FSMD $M_2$. Both of these FSMDs compute the greatest common divisor of two numbers. The states $\{q_{1,0}, q_{1,1}, q_{1,2}, q_{1,3}, q_{1,4}, q_{1,5}\}$ for $M_1$ and the states $\{q_{2,0}, q_{2,1}\}$ for $M_2$ are initially chosen as the cut-points. We use the notation $q_i \twoheadrightarrow q_j$ to represent a path from the state $q_i$ to the state $q_j$. To distinguish between paths originating from the same state due to conditional branches, we write $q_i \xrightarrow{c}\!\!\!\!\twoheadrightarrow q_j$ to denote the path from $q_i$ to $q_j$ which is traversed when the condition $c$ is satisfied. Note that

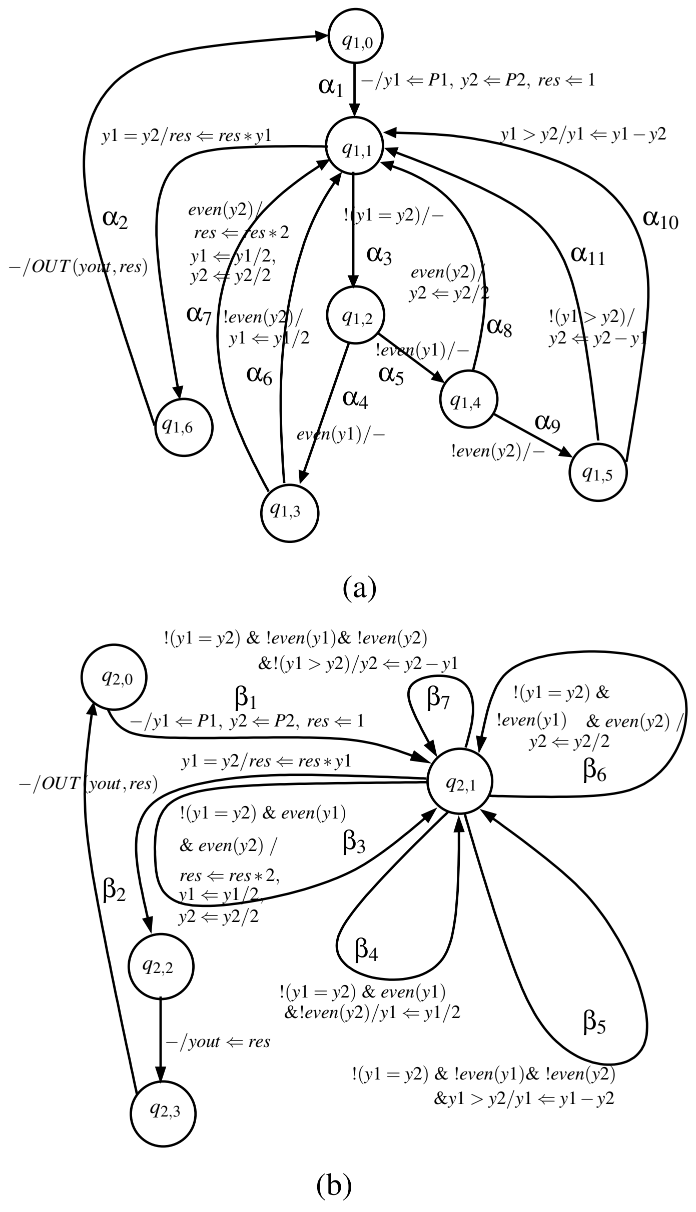


Figure 3.1: (a) $M_1$: Source FSMD. (b) $M_2$: Transformed FSMD.

upon finding a mismatch between a pair of paths, the one with the weaker condition of execution is extended. The algorithm reported in [95] proceeds in the following sequence:

1) finds $\beta_1$ as the equivalent path of $\alpha_1$;
2) finds $\beta_2$ as the equivalent path of $\alpha_2$;
3) fails to find equivalent path of $\alpha_3$, hence, extends it; the extended paths are $\alpha_3\alpha_4$ and $\alpha_3\alpha_5$;
4) fails to find equivalent path of $\alpha_3\alpha_4$, hence, extends it; the extended paths are $\alpha_3\alpha_4\alpha_6$ and $\alpha_3\alpha_4\alpha_7$;
5) fails to find equivalent path of $\alpha_3\alpha_5$, hence, extends it; the extended paths are $\alpha_3\alpha_5\alpha_8$ and $\alpha_3\alpha_5\alpha_9$;
6) and 7) finds $\beta_4$ and $\beta_3$ as the respective equivalent paths of $\alpha_3\alpha_4\alpha_6$ and $\alpha_3\alpha_4\alpha_7$;
8) finds $\beta_6$ as the equivalent path of $\alpha_3\alpha_5\alpha_8$;
9) fails to find equivalent path of $\alpha_3\alpha_5\alpha_9$, hence, extends it; the extended paths are $\alpha_3\alpha_5\alpha_9\alpha_{10}$ and $\alpha_3\alpha_5\alpha_9\alpha_{11}$;
10) and 11) find $\beta_5$ and $\beta_7$ as the respective equivalent path of $\alpha_3\alpha_5\alpha_9\alpha_{10}$ and $\alpha_3\alpha_5\alpha_9$ $\alpha_{11}$.

Thus, the set of corresponding states for this example is $\{\langle q_{1,0}, q_{2,0}\rangle, \langle q_{1,1}, q_{2,1}\rangle\}$. It follows from Theorem 1, $M_1 \sqsubseteq M_2$; on reversing the roles of $M_1$ and $M_2$, again it follows from Theorem 1, $M_2 \sqsubseteq M_1$. Hence, we conclude that $M_1$ and $M_2$ are equivalent from Definition 3. The respective path covers of the source and the transformed FSMDs such that there is a one-to-one correspondence between their members in terms of path equivalence are as follows:

$$
\begin{aligned}
P_1 = \{\ & q_{1,0} \twoheadrightarrow q_{1,1}, \\
& q_{1,1} \xrightarrow{y1=y2} q_{1,0}, \\
& q_{1,1} \xrightarrow{!(y1=y2)} q_{1,2} \xrightarrow{even(y1)} q_{1,3} \xrightarrow{even(y2)} q_{1,1}, \\
& q_{1,1} \xrightarrow{!(y1=y2)} q_{1,2} \xrightarrow{even(y1)} q_{1,3} \xrightarrow{!even(y2)} q_{1,1}, \\
& q_{1,1} \xrightarrow{!(y1=y2)} q_{1,2} \xrightarrow{!even(y1)} q_{1,4} \xrightarrow{even(y2)} q_{1,1}, \\
& q_{1,1} \xrightarrow{!(y1=y2)} q_{1,2} \xrightarrow{!even(y1)} q_{1,4} \xrightarrow{!even(y2)} q_{1,5} \xrightarrow{y1>y2} q_{1,1}, \\
& q_{1,1} \xrightarrow{!(y1=y2)} q_{1,2} \xrightarrow{!even(y1)} q_{1,4} \xrightarrow{!even(y2)} q_{1,5} \xrightarrow{!(y1>y2)} q_{1,1}\ \}, \\
P_2 = \{\ & q_{2,0} \twoheadrightarrow q_{2,1}, \\
& q_{2,1} \xrightarrow{y1=y2} q_{2,0}, \\
& q_{2,1} \xrightarrow{!(y1=y2)\&even(y1)\&even(y2)} q_{2,1}, \\
& q_{2,1} \xrightarrow{!(y1=y2)\&even(y1)\&!even(y2)} q_{2,1}, \\
& q_{2,1} \xrightarrow{!(y1=y2)\&!even(y1)\&even(y2)} q_{2,1}, \\
& q_{2,1} \xrightarrow{!(y1=y2)\&!even(y1)\&!even(y2)\&y1>y2} q_{2,1}, \\
& q_{2,1} \xrightarrow{!(y1=y2)\&!even(y1)\&!even(y2)\&!(y1>y2)} q_{2,1}\ \}.
\end{aligned}
$$

■

The path extension based approaches [90, 95, 110] do modify the initial set of path covers to new sets (as illustrated through the above example) so that the resulting path covers satisfy the property characterized in Theorem 1; they, however, cannot validate code motions across loops. In the subsequent section, we devise a method whereby an established path based approach is moulded suitably to verify such transformations. In this work, a path cover is obtained by setting the reset state and the branching states (i.e., states with more than one outward transition) of the FSMD as cut-points.

## 3.3 The method of symbolic value propagation

### 3.3.1 Basic concepts

The symbolic value propagation method consists in propagating values of variables over the corresponding paths of the two FSMDs on discovery of mismatch in the values of some live variable; the mismatched values are marked to distinguish them from the matched values. A variable $v$ is said to be *live* at a state $s$ if there is an execution sequence starting at $s$ along which its value may be used before $v$ is redefined [10]. Propagation of values from a path $\alpha_1$ to the next path $\alpha_2$ is accomplished by associating a *propagated vector* at the end state of the path $\alpha_1$ (or equivalently, the start state of the path $\alpha_2$). A *propagated vector* $\overline{\vartheta}$ through a path $\alpha_1$ is an ordered pair of the form $\langle \mathcal{C}, \langle e_1, e_2, \cdots, e_k \rangle \rangle$, where $k = |V_1 \bigcup V_2|$. The first element $\mathcal{C}$ of the pair represents the condition that has to be satisfied at the start state of $\alpha_1$ to traverse the path and reach its end state with the upgraded propagated vector. The second element, referred to as *value-vector*, comprises of $e_i$, $1 \leq i \leq k$, which represents the symbolic value attained at the end state of $\alpha_1$ by the variable $v_i \in V_1 \bigcup V_2$. To start with, for the reset state, the propagated vector is $\langle \top, \langle v_1, v_2, \cdots, v_k \rangle \rangle$ (also represented as $\overline{\upsilon}$), where $\top$ stands for *true* and $e_i = v_i, 1 \leq i \leq k$, indicates that the variables are yet to be defined. For brevity, we represent the second component as $\overline{v}$ to mean $\langle v_1, v_2, \cdots, v_k \rangle$ and as $\overline{e}$ to mean $\langle e_1, e_2, \cdots, e_k \rangle$, where $e_i$'s are the symbolic expressions (values) involving variables $v_i, 1 \leq i \leq k$, in general. It is important to note that an uncommon variable, i.e., a variable that is defined in either of the FSMDs but not both, from the set $V_2 - V_1$ $(V_1 - V_2)$ retains its symbolic value in the propagated vectors of $M_1$ $(M_2)$ throughout a computation of $M_1$ $(M_2)$. Let $R_\alpha(\overline{v})$ and $s_\alpha(\overline{v})$ represent respectively the condition of execution and the data transformation of a path $\alpha$ when there is no propagated vector at the start state $\alpha^s$ of $\alpha$. In the presence of a propagated vector, $\overline{\vartheta_{\alpha^s}} = \langle c_1, \overline{e} \rangle$ say, at $\alpha^s$, its condition of execution becomes $c_1(\overline{v}) \wedge R_\alpha(\overline{v})\{\overline{e}/\overline{v}\}$ and the data transformation becomes $s_\alpha(\overline{v})\{\overline{e}/\overline{v}\}$, where $\{\overline{e}/\overline{v}\}$ is called a substitution; the expression $\kappa\{\overline{e}/\overline{v}\}$ represents that each variable $v_j \in \overline{v}$ that occurs in $\kappa$ is replaced by the corresponding expression $e_j \in \overline{e}$ simultaneously with other variables. The propagated vector $\overline{\vartheta_\alpha}$ associated with a path $\alpha : \alpha^s \twoheadrightarrow \alpha^f$ will synonymously be referred to as $\overline{\vartheta_{\alpha^f}}$ associated with the final state $\alpha^f$ of $\alpha$. Also, we use the symbol $\overline{\vartheta_{i,j}}$ to represent a propagated vector corresponding to the state $q_{i,j}$. We follow a similar convention (of suffixing) for the members of the propagated vector, namely the condition and the value-vector. The above discussion provides the foundation for comparing two paths, and hence the following definition is in order.

**Definition 7** (Equivalent paths for a pair of propagated vectors)**.** *A path* $\alpha : q_{1,s} \twoheadrightarrow q_{1,f}$ *of FSMD* $M_1$ *with a propagated vector* $\langle \mathcal{C}_\alpha, \overline{v_{1,s}} \rangle$ *is said to be (unconditionally) equivalent (denoted using* $\simeq$*) to a path* $\beta : q_{2,s} \twoheadrightarrow q_{2,f}$ *of FSMD* $M_2$ *with a propagated vector* $\langle \mathcal{C}_\beta, \overline{v_{2,s}} \rangle$ *if* $\mathcal{C}_\alpha \wedge R_\alpha(\overline{v_{1,s}}/\bar{v}) \equiv \mathcal{C}_\beta \wedge R_\beta(\overline{v_{2,s}}/\bar{v})$ *and* $\mathsf{s}_\alpha(\overline{v_{1,s}}/\bar{v}) = \mathsf{s}_\beta(\overline{v_{2,s}}/\bar{v})$*. Otherwise, the path* $\alpha$ *with the above propagated vector is said to be conditionally equivalent (denoted using* $\simeq_c$*) to the path* $\beta$ *with the corresponding propagated vector if* $q_{1,f} \neq q_{1,0}$*, the reset state of* $M_1$*,* $q_{2,f} \neq q_{2,0}$*, the reset state of* $M_2$ *and* $\forall \alpha'$ *emanating from the state* $q_{1,f}$ *with the propagated vector* $\langle \mathcal{C}_\alpha \wedge R_\alpha(\overline{v_{1,s}}/\bar{v}), \mathsf{s}_\alpha(\overline{v_{1,s}}/\bar{v}) \rangle$*,* $\exists \beta'$ *from* $q_{2,f}$ *with the propagated vector* $\langle \mathcal{C}_\beta \wedge R_\beta(\overline{v_{2,s}}/\bar{v}), \mathsf{s}_\beta(\overline{v_{2,s}}/\bar{v}) \rangle$*, such that* $\alpha' \simeq \beta'$ *or* $\alpha' \simeq_c \beta'$*.*

In Definition 7, the conditions $q_{1,f} \neq q_{1,0}$ and $q_{2,f} \neq q_{2,0}$ prevent symbolic value propagation beyond the reset states. It also implies that if $\alpha'$ terminates in $q_{1,0}$ such that it has an equivalent path $\beta'$, then $\alpha' \simeq \beta'$ must hold; more specifically, paths terminating in reset states cannot have any conditionally equivalent paths. To distinguish explicitly between *conditionally equivalent* paths and *unconditionally equivalent* paths, the terms *C-equivalent* and *U-equivalent* are henceforth used. The final states of two C-equivalent paths are called *conditionally corresponding (C-corresponding) states*.

**Example 2.** Figure 3.2 illustrates the method of symbolic value propagation. Let the variable ordering be $\langle u, v, w, x, y, z \rangle$. The states $q_{1,0}$ and $q_{2,0}$ are the reset states of the FSMDs that are being checked for equivalence. The propagated vector corresponding to each reset state is $\overline{\upsilon}$. The propagated vectors at the end states $q_{1,1}$ and $q_{2,1}$ of the paths $\alpha_1$ and $\beta_1$ respectively are found to be $\overline{\vartheta_{1,1}} = \langle \top, \langle u, v, w, \mathbf{f_1}(\mathbf{u}, \mathbf{v}), \mathbf{y}, z \rangle \rangle$ and $\overline{\vartheta_{2,1}} = \langle \top, \langle u, v, w, \mathbf{x}, \mathbf{f_2}(\mathbf{u}), z \rangle \rangle$, respectively. The variables $u$, $v$, $w$ and $z$ did not have any propagated value at the beginning and have not changed their values along the paths. The values of $x$ and $y$ are given in bold face to denote that they mismatch; when the value of some variables is propagated in spite of a match (this is done to resolve transformations such as copy propagation), they are given in normal face. The mismatched values are demarcated because they require special treatment during analysis of code motion across loop, as explained shortly in Section 3.3.2. Next we use the characteristics of the paths $\alpha_2$ and $\beta_2$ to obtain the respective conditions of execution and data transformations for these propagated vectors. Let $\Pi_i^n$ represent the

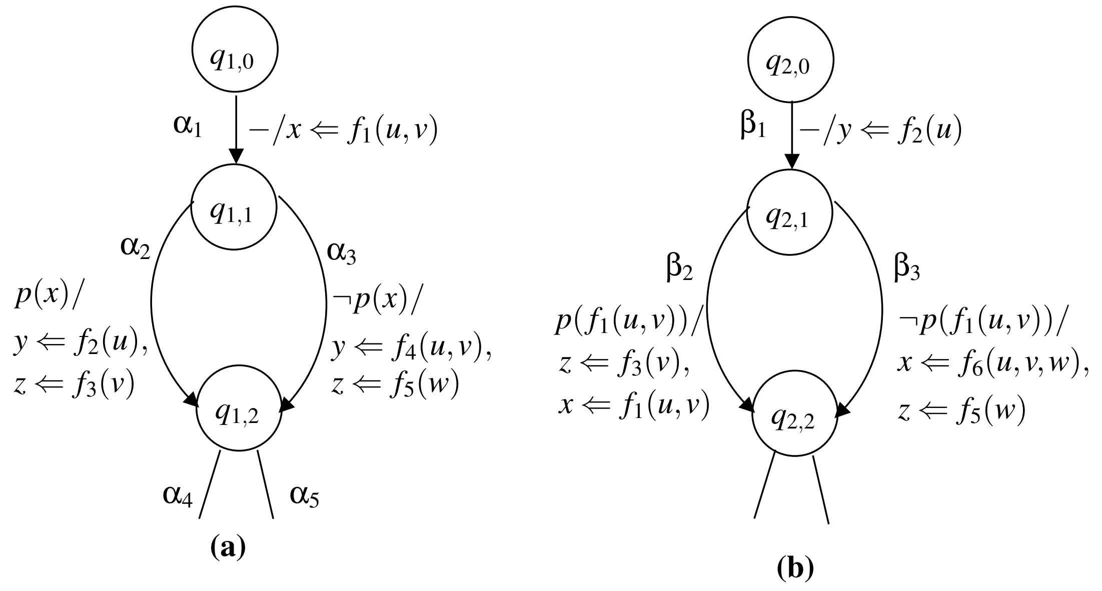


Figure 3.2: An example of symbolic value propagation.

$i$-th projection function of arity $n$; we keep the arity understood when it is clear from the context. For the path $\alpha_2$, note that the condition of execution $R_{\alpha_2}$ is $p(x)$ and the data transformation $s_{\alpha_2}$ is $\{y \Leftarrow f_2(u), z \Leftarrow f_3(v)\}$. The modified condition of execution, $R'_{\alpha_2}$ is calculated as $\Pi_1(\overline{\vartheta_{1,1}}) \wedge R_{\alpha_2}(\overline{v})\{\Pi_2(\overline{\vartheta_{1,1}})/\overline{v}\} \equiv \top \wedge p(x)\{f_1(u,v)/x\} \equiv p(f_1(u,v))$ and the modified data transformation $s'_{\alpha_2}$ as $s_{\alpha_2}(\overline{v})\{\Pi_2(\overline{\vartheta_{1,1}})/\overline{v}\} = \{x \Leftarrow f_1(u,v), y \Leftarrow f_2(u), z \Leftarrow f_3(v)\}$. The symbol $s'_{\alpha_2}\big|_x$ represents the value corresponding to the variable $x$ as transformed after application of $s_{\alpha_2}$ on the propagated vector $\Pi_2(\overline{\vartheta_{1,1}})$, i.e., $f_1(u,v)$. The characteristic tuple for path $\beta_2$ comprises $R_{\beta_2}(\overline{v}) \equiv p(f_1(u,v))$ and $s_{\beta_2}(\overline{v}) = \{z \Leftarrow f_3(v), x \Leftarrow f_1(u,v)\}$. The condition $R'_{\beta_2}$ of the path $\beta_2$ and the data transformation $s'_{\beta_2}$ of the propagated vector are calculated similarly as $R'_{\beta_2} \equiv \Pi_1(\overline{\vartheta_{2,1}}) \wedge R_{\beta_2}(\overline{v})\{\Pi_2(\overline{\vartheta_{2,1}})/\overline{v}\} \equiv \top \wedge p(f_1(u,v)) \equiv p(f_1(u,v))$ and $s'_{\beta_2} = s_{\beta_2}(\overline{v})\{\Pi_2(\overline{\vartheta_{2,1}})/\overline{v}\} = \{x \Leftarrow f_1(u,v), y \Leftarrow f_2(u), z \Leftarrow f_3(v)\}$. We find that $R'_{\alpha_2} \equiv R'_{\beta_2}$ and $s'_{\alpha_2} = s'_{\beta_2}$. Since a match in the characteristic tuples of the respective paths implies symbolic value propagation is no longer required in this case, there is no need to store the vectors; the paths $\alpha_2$ and $\beta_2$ are declared as U-equivalent.

For the paths $\alpha_3$ and $\beta_3$, having the conditions of execution $R_{\alpha_3} \equiv \neg p(x)$ and $R_{\beta_3} \equiv \neg p(f_1(u,v))$ respectively, the respective conditions $R'_{\alpha_3}$ and $R'_{\beta_3}$ with respect to $\overline{\vartheta_{\alpha_3^s}}$ and $\overline{\vartheta_{\beta_3^s}}$ are found to be $\neg p(f_1(u,v))$. The data transformations are $s_{\alpha_3} = \{y \Leftarrow f_4(u,v), z \Leftarrow f_5(w)\}$ and $s_{\beta_3} = \{x \Leftarrow f_6(u,v,w), z \Leftarrow f_5(w)\}$. The respective modified data transformations are $s'_{\alpha_3} = \{x \Leftarrow f_1(u,v), y \Leftarrow f_4(u,v), z \Leftarrow f_5(w)\}$ and

**Algorithm 1** valuePropagation ($\alpha$, $\langle C_{\alpha^s}, \overline{v_{\alpha^s}}\rangle$, $\beta$, $\langle C_{\beta^s}, \overline{v_{\beta^s}}\rangle$)

**Inputs:** Two paths $\alpha$ and $\beta$, and two propagated vectors $\langle C_{\alpha^s}, \overline{v_{\alpha^s}}\rangle$ at $\alpha^s$ and $\langle C_{\beta^s}, \overline{v_{\beta^s}}\rangle$ at $\beta^s$.

**Outputs:** Propagated vector $\langle C_{\alpha^f}, \overline{v_{\alpha^f}}\rangle$ for $\alpha^f$ and propagated vector $\langle C_{\beta^f}, \overline{v_{\beta^f}}\rangle$ for $\beta^f$.

```
1: $C_{\alpha^f} \leftarrow C_{\alpha^s} \wedge R_\alpha(\overline{v})\{\overline{v_{\alpha^s}}/\overline{v}\}$;  $C_{\beta^f} \leftarrow C_{\beta^s} \wedge R_\beta(\overline{v})\{\overline{v_{\beta^s}}/\overline{v}\}$.
2: if $\exists$ variable $v_i \in (V_1 \bigcup V_2)$ which is live at $\alpha^f$ or $\beta^f$
       and $s_\alpha(\overline{v})\{\overline{v_{\alpha^s}}/\overline{v}\}\Big|_{v_i} \neq s_\beta(\overline{v})\{\overline{v_{\beta^s}}/\overline{v}\}\Big|_{v_i}$ then
3:    $\forall\, v_j \in (V_1 \bigcup V_2)$, $\Pi_j(\overline{v_{\alpha^f}}) \leftarrow s_\alpha(\overline{v})\{\overline{v_{\alpha^s}}/\overline{v}\}\Big|_{v_j}$.
4:    $\forall\, v_j \in (V_1 \bigcup V_2)$, $\Pi_j(\overline{v_{\beta^f}}) \leftarrow s_\beta(\overline{v})\{\overline{v_{\beta^s}}/\overline{v}\}\Big|_{v_j}$.
5:    Mark each variable, $x$ say, which exhibits mismatch at $\alpha^f$ and $\beta^f$, and also mark all
      those variables on which $x$ depends.
6: end if
7: return $\langle\langle C_{\alpha^f}, \overline{v_{\alpha^f}}\rangle, \langle C_{\beta^f}, \overline{v_{\beta^f}}\rangle\rangle$.
```

$s'_{\beta_3} = \{x \Leftarrow f_6(u,v,w), y \Leftarrow f_2(u), z \Leftarrow f_5(w)\}$. There is a mismatch between $s'_{\alpha_3}$ and $s'_{\beta_3}$ for the values of $x$ and $y$; hence the propagated vectors that are stored at $q_{1,2}$ and $q_{2,2}$ (via $\alpha_3$ and $\beta_3$) are $\overline{\vartheta_{1,2}} = \langle \neg p(f_1(u,v)), \langle u,v,w,\mathbf{f_1}(\mathbf{u},\mathbf{v}),\mathbf{f_4}(\mathbf{u},\mathbf{v}), f_5(w)\rangle\rangle$ and $\overline{\vartheta_{2,2}} = \langle \neg p(f_1(u,v)), \langle u,v,w,\mathbf{f_6}(\mathbf{u},\mathbf{v},\mathbf{w}),\mathbf{f_2}(\mathbf{u}), f_5(w)\rangle\rangle$ respectively. In this example, the propagated vectors $\overline{\vartheta_{1,1}}$ at $q_{1,1}$ and $\overline{\vartheta_{2,1}}$ at $q_{2,1}$ got identically transformed over paths $\alpha_2$ and $\beta_2$, respectively. Had they not matched we would have to store the corresponding propagated vectors along these paths also. This indicates that we may have to store more than one propagated vector in any state. This, however, will not be the case. Owing to the depth-first traversal approach achieved through recursive invocations, it is sufficient to work with a single propagated vector at a time for each state. Suppose we first reach the state $q_{1,2}$ via the path $\alpha_2$ with the mismatched propagated vector $\overline{\vartheta_{\alpha_2^f}}$. Only after the U-equivalent or C-equivalent of each of the paths $\alpha_4$ and $\alpha_5$ emanating from $q_{1,2}$ using $\overline{\vartheta_{\alpha_2^f}}$ are found, does the procedure again visit the state $q_{1,2}$ corresponding to $\alpha_3$. Finally, when $\alpha_3$ is accounted for, we store simply $\overline{\vartheta_{\alpha_3^f}}$ in $q_{1,2}$. ∎

It has been stated above that only the values of those live variables which exhibit mismatch are marked in the propagated vectors; however, those variables on which these mismatched variables depend need to be marked as well to detect valid code motion across loop as borne out by the example in Section 3.3.2. The function valuePropagation (Algorithm 1) formalizes the above steps of computation of the propagated vectors in case of a mismatch.

Using symbolic value propagation as a pivotal concept, we need to formalize our main task of equivalence checking of two given FSMDs. However, certain intricacies are interlaced with symbolic value propagation which need to be resolved first. The following subsections illustrate these intricacies and furnishes the mechanisms for addressing them. The overall verification method is then presented in a subsequent section.

### 3.3.2 Need for detecting loop invariance of subexpressions

In order to verify behaviours in the presence of code motions beyond loops, one needs to ascertain whether certain subexpressions remain invariants in a loop or not. Basically, suppose some variable, $w$ say, gets defined before a loop $L$ in the original behaviour, and *somehow* it can be confirmed that the value of $w$ remains invariant in that loop. Now, if an equivalent definition for $w$ is found in the other behaviour after exiting the loop $L'$ (which corresponds to $L$), then such behaviours will indeed be equivalent. In this work, we target to establish the loop invariance of propagated vectors which, in turn, suffices to prove the loop invariance of the involved variables. The following example is used to reveal these facts more vividly.

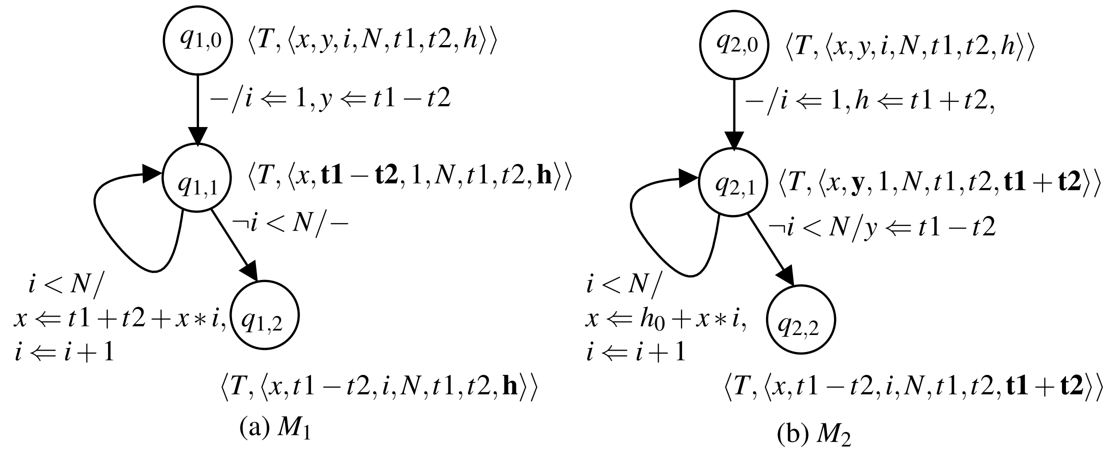

$q_{1,0}$ $\langle T, \langle x,y,i,N,t1,t2,h\rangle\rangle$
$-/i \Leftarrow 1, y \Leftarrow t1-t2$
$q_{1,1}$ $\langle T, \langle x,\mathbf{t1-t2},1,N,t1,t2,\mathbf{h}\rangle\rangle$
$\neg i < N/-$
$i<N/$ $x \Leftarrow t1+t2+x*i,$ $i \Leftarrow i+1$
$q_{1,2}$
$\langle T, \langle x,t1-t2,i,N,t1,t2,\mathbf{h}\rangle\rangle$
(a) $M_1$
$q_{2,0}$ $\langle T, \langle x,y,i,N,t1,t2,h\rangle\rangle$
$-/i \Leftarrow 1, h \Leftarrow t1+t2,$
$q_{2,1}$ $\langle T, \langle x,\mathbf{y},1,N,t1,t2,\mathbf{t1+t2}\rangle\rangle$
$\neg i < N/y \Leftarrow t1-t2$
$i<N/$ $x \Leftarrow h_0+x*i,$ $i \Leftarrow i+1$
$q_{2,2}$
$\langle T, \langle x,t1-t2,i,N,t1,t2,\mathbf{t1+t2}\rangle\rangle$
(b) $M_2$

Figure 3.3: An example of propagation of values across loop.

**Example 3.** In Figure 3.3, the operation $y \Leftarrow t1-t2$, which is originally placed before the loop body in $M_1$ (Figure 3.3(a)), is moved after the loop in the transformed FSMD $M_2$ (Figure 3.3(b)). This transformation reduces the register lifetime for $y$. In addition, the subexpression $t1+t2$, which is originally computed in every iteration of the loop, is now computed only once before the start of the loop and the result is stored in an uncommon variable $h$ which, in turn, is used in every loop iteration. These two transformations are possible because the variables $t1$, $t2$ are not updated and $y$ is neither used nor updated within the loop body.

Now consider the equivalence checking between these two behaviours. The path $q_{1,0} \twoheadrightarrow q_{1,1}$ of $M_1$ is said to be candidate C-equivalent to the path $q_{2,0} \twoheadrightarrow q_{2,1}$ of $M_2$ since the values of $y$ and $h$ mismatch. Accordingly, we have the propagated value $t1 - t2$ for $y$ at $q_{1,1}$ and $t1 + t2$ for $h$ at $q_{2,1}$. Now, the path $q_{1,1} \twoheadrightarrow q_{1,1}$ (which represents a loop) is declared to be candidate C-equivalent of $q_{2,1} \twoheadrightarrow q_{2,1}$ with the propagated values of $h$ and $y$. The variables $y$ and $h$ are not updated in either of the loop bodies; nor are the variables $t1$, $t2$ participating in the expression values $t1 - t2$ and $t1 + t2$ of $y$ and $h$, respectively. Hence, we have the same propagated values at $q_{1,1}$ and $q_{2,1}$ after executing the loops indicating that code motion across loop may have taken place. Since $h$ is an uncommon variable (in $V_2 - V_1$), its value can never match over the two FSMDs. Therefore, if a definition of the common variable $y$ is detected after the loop in $M_2$ which renders the values of $y$ in the two FSMDs equivalent, then such code motions across the loop would indeed be valid. In the current example, this occurs in the path $q_{2,1} \twoheadrightarrow q_{2,2}$ and hence finally, $q_{1,1} \twoheadrightarrow q_{1,2}$ and $q_{2,1} \twoheadrightarrow q_{2,2}$ are designated as U-equivalent paths with matches in all the variables except in the uncommon variable, and the previously declared candidate C-equivalent path pairs are asserted to be actually C-equivalent. ■

The above example highlights the intricacy involved in verification of code motions across loops, that is, detection of loop invariant subexpressions. Example 3 exhibits only simple loops. However, a loop may comprise of several paths because of branchings contained in a loop. The conventional approach of using back edges[1] to detect loops has been used in [21]. This approach however fails in the presence of dynamic loop scheduling (DLS) [135]. In Section 3.5.2, an example of DLS is presented and its equivalence is established by our verification procedure with the help of a different loop detection mechanism as described below.

To identify loops, a *LIST* of (candidate) C-equivalent pairs of paths is maintained. Initially, the *LIST* is empty. Whenever a pair of (candidate) C-equivalent paths is detected, they are added to the *LIST*. Thus, if a path occurs in the *LIST*, then it indicates that the final state of the path has an associated propagated vector with some

[1]A back edge $(u, v)$ connects $u$ to an ancestor $v$ in a depth-first tree of a directed graph [42].

unresolved mismatch(es). If at any point of symbolic value propagation, a path with the start state $q$ is encountered which is also the final state of some path appearing in *LIST*, it means that a loop has been encountered with the mismatch that was identified during the last visit of $q$ (stored as a propagated vector) not yet resolved[2]. Once pairs of U-equivalent paths are subsequently detected for every path emanating from the states $\alpha^f$ and $\beta^f$ of a candidate C-equivalent path pair $\langle\alpha,\beta\rangle$ present in the *LIST*, the paths $\alpha$ and $\beta$ are declared as *actually* C-equivalent (by Definition 7).

If a loop is crossed over with some mismatches persisting, it means that the last constituent path of the loop is not yet U-equivalent. Let $q$ be both the entry and exit state of such a loop; the propagated vector already stored at $q$ (during entry) and the one computed after traversal of the final path of the loop in $M$ leading to $q$ need to be compared. This is explained with the help of Example 3. Let us first consider a variable, such as $y$ in the example, whose values mismatched while entering the loop. Now, if this variable had got updated within the loop, then surely it was not an invariant in the loop, a fact which would have easily been detected by the presence of a mismatch for $y$ between the initial and the final vectors of $q_{1,1} \twoheadrightarrow q_{1,1}$ or $q_{2,1} \twoheadrightarrow q_{2,1}$. Secondly, consider those variables on which the mismatched variables depend such as, $t1$ and $t2$. If any of them were to get modified within the loop (even if identically for both the FSMDs), then moving the operation $y \Leftarrow t1 - t2$ from the path $q_{1,0} \twoheadrightarrow q_{1,1}$ in $M_1$ to the path $q_{2,1} \twoheadrightarrow q_{2,2}$ in $M_2$ would not have been a valid case of code motion across loop. Such modifications (in $t1$ or $t2$) also get detected by comparing the vectors since we mark the value of those variables as well on which some mismatched variable depends (as was mentioned in Section 3.3) in the propagated vector. Lastly, all other variables should get defined identically in the two loops for their data transformations to match. If any difference in the value of some marked variable between entry and exit to a loop is detected, then we can ascertain that this variable is not an invariant in the loop. It further implies that the code motion across the loop need not be valid. Since it cannot be determined statically how many times a loop will iterate before exiting, it is not possible to determine what values will the unmarked variables take when a loop iterates. Moreover, the unmarked variables also do not participate in code motion across loop, hence they are reverted to their symbolic values at the exit of a loop. For example, the variable $i$ in both the FSMDs of Figure 3.3 has the respective

[2]While treating the example in Section 3.5.2, we underline a situation where the loop detection mechanism based on back edge fails and the above method succeeds.

values 1 and 2 during the entry and the exit of the loop. However, before the paths $q_{1,1} \twoheadrightarrow q_{1,2}$ and $q_{2,1} \twoheadrightarrow q_{2,2}$ are compared, the value of $i$ is set to its symbolic value "$i$" (instead of 2) in the propagated vectors computed at $q_{1,1}$ and $q_{2,1}$ after the respective loops are exited. The function loopInvariant (Algorithm 2) accomplishes these tasks.

**Algorithm 2** loopInvariant ($\gamma$, $\overline{\vartheta'_{\gamma^f}}$, $\overline{\vartheta_{\gamma^f}}$, $\overline{\vartheta'_{\varsigma^f}}$)

**Inputs:** A path $\gamma$, a propagated vector $\overline{\vartheta'_{\gamma^f}}$ which is computed after $\gamma$, the propagated vector $\overline{\vartheta_{\gamma^f}}$ stored in the cut-point $\gamma^f$, which is the entry/exit state of a loop, and the propagated vector $\overline{\vartheta'_{\varsigma^f}}$ where $\varsigma$ is the C-corresponding path of $\gamma$.

**Outputs:** A Boolean value.

```
1: if $\exists$ a marked variable $x$, $\overline{\vartheta'_{\gamma^f}}\Big|_x \neq \overline{\vartheta_{\gamma^f}}\Big|_x$ then
2:     return false.
3: else if $\exists$ an unmarked variable $x$, $\overline{\vartheta'_{\gamma^f}}\Big|_x \neq \overline{\vartheta'_{\varsigma^f}}\Big|_x$ then
4:     return false.
5: end if
6: Set each unmarked variable $x$ to its symbolic value "$x$".
7: return true.
```

### 3.3.3 Subsumption of conditions of execution of the paths being compared

It is worth noting that the examples demonstrated in Figure 3.2 and Figure 3.3 illustrate the case $R_\alpha \equiv R_\beta$. Another intricacy arises when the condition $R_\alpha \not\equiv R_\beta$ but $R_\alpha \Rightarrow R_\beta$ or $R_\beta \Rightarrow R_\alpha$ is encountered. Specifically, to handle this situation, the notion of *null path* is introduced in an FSMD, to force symbolic value propagation along the path in the other FSMD. A null path (of length 0) from any state $q$ to the same state $q$ has the condition of execution $\top$ and a null (identity) data transformation. We refer to a null path emanating from a state $q$ as $\varpi_q$ having the start state $\varpi_q^s$ same as the final state $\varpi_q^f = q$. The example given below elucidates on how to handle these cases.

**Example 4.** In Figure 3.4, let the states $q_{1,r}$ and $q_{2,k}$ be the corresponding states and the variable ordering be $\langle u,v,w,x,y\rangle$. When we search for a path starting from $q_{2,k}$ that is C-equivalent to $\alpha_1$, we find $R_{\alpha_1} \Rightarrow R_{\beta_1}$ (i.e. $c \Rightarrow \top$). The path $\beta_1$ is compared with $\varpi_{q_{1,r}}$; in the process, the vector propagated to $q_{2,l}$ becomes $\overline{\vartheta_{2,l}} = \langle \top, \langle u,v,w,\mathbf{f}(\mathbf{u},\mathbf{v}),y\rangle\rangle$ (and $\varpi_{q_{1,r}}$ is held C-equivalent to $\beta_1$). For the path $\beta_2$, $R_{\beta_2} \equiv c$

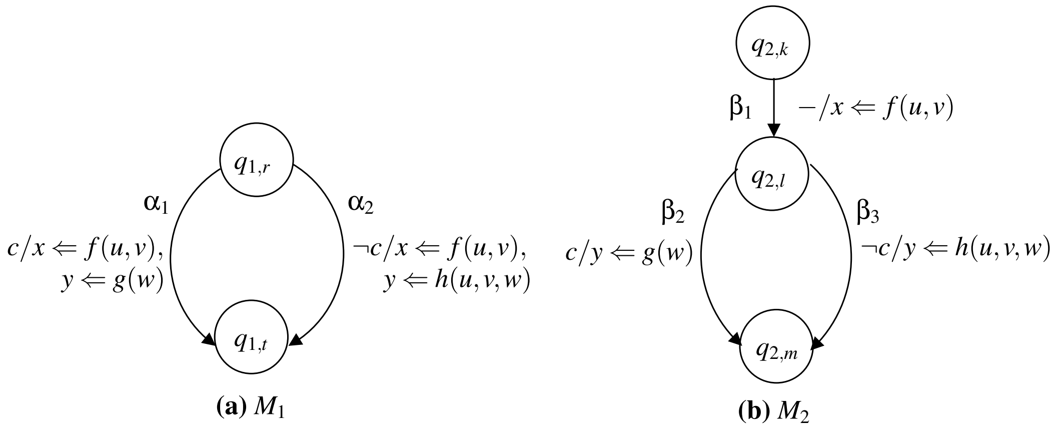


Figure 3.4: An example to illustrate the cases $R_\alpha \Rightarrow R_\beta$ and $R_\beta \Rightarrow R_\alpha$.

and $s_{\beta_2} = \{y \Leftarrow g(w)\}$. Based on $\overline{\vartheta_{2,l}}$, the path characteristics are then calculated to be $R'_{\beta_2} \equiv c$ and $s'_{\beta_2} = \{x \Leftarrow f(u,v), y \Leftarrow g(w)\}$ which are equivalent to that of $\alpha_1$. Analogously, the path characteristics of $\alpha_2$ and $\beta_3$ (with $\overline{\vartheta_{2,l}}$) are found to be equivalent.

When we have to show the converse, i.e., $M_2 \sqsubseteq M_1$, we are required to find a path in $M_1$ that is equivalent to $\beta_1$. We find two paths $\alpha_1$ and $\alpha_2$ such that $R_{\alpha_1} \Rightarrow R_{\beta_1}$ (i.e. $c \Rightarrow \top$) and also $R_{\alpha_2} \Rightarrow R_{\beta_1}$ (i.e. $\neg c \Rightarrow \top$). The vector $\overline{\vartheta_{2,l}}$ (same as above) is propagated to $q_{2,l}$ after comparing $\beta_1$ with $\varpi_{q_{1,r}}$. Then we recursively try to find the paths equivalent to $\beta_2$ and $\beta_3$, thereby obtaining the set of C-equivalent paths: $\{\langle \beta_1, \varpi_{q_{1,r}} \rangle\}$ and the set of U-equivalent paths: $\{\langle \beta_2, \alpha_1 \rangle, \langle \beta_3, \alpha_2 \rangle\}$. ■

To summarize, therefore, of the paths $\alpha$ and $\beta$, the one having a stronger condition will have a null path introduced at its initial state to force symbolic value propagation along the other path. The function findEquivalentPath (Algorithm 3) seeks to find a pair of U-equivalent or C-equivalent paths depending on how the conditions of execution of the paths being compared are related to each other. The propagated vectors for the end states of these paths also get updated accordingly.

Specifically, the function findEquivalentPath takes as inputs a path $\alpha$ of the FSMD $M_1$, a propagated vector $\overline{\vartheta_{\alpha^s}}$, a state $q_{2,j}$ of the FSMD $M_2$, which has correspondence with the state $\alpha^s$, a propagated vector $\overline{\vartheta_{\beta^s}}$ at $q_{2,j}$, and the path covers $P_1$ and $P_2$ of the FSMDs $M_1$ and $M_2$, respectively. The function's objective is to find a U-equivalent or C-equivalent path for $\alpha$ (with respect to $\overline{\vartheta_{\alpha^s}}$) in $P_2$ emanating from $q_{2,j}$. For the paths in $P_2$ emanating from $q_{2,j}$, the characteristic tuples are computed with respect to $\overline{\vartheta_{\beta^s}}$. It returns a 4-tuple $\langle \alpha_m, \beta, \overline{\vartheta_{\alpha_m^f}}, \overline{\vartheta_{\beta^f}} \rangle$, which is defined depending upon the following cases. In all the cases, $\alpha_m$ is a path in $P_1$, $\beta$ is a path in $P_2$, $\overline{\vartheta_{\alpha_m^f}}$ is the propagated vector at the end state $\alpha_m^f$ of $\alpha_m$, and $\overline{\vartheta_{\beta^f}}$ is the propagated vector at the end state $\beta^f$ of $\beta$. In the algorithm, the symbols $R'_\alpha$ and $R'_\beta$ denote respectively the condition of execution of the path $\alpha$ with respect to $\overline{\vartheta_{\alpha^s}}$ and that of the path $\beta$ with respect to $\overline{\vartheta_{\beta^s}}$; likewise for $\mathsf{s}'_\alpha$ and $\mathsf{s}'_\beta$. We have the following cases:

*Case 1* ($R'_\alpha \equiv R'_\beta$): A path $\beta$ in $P_2$ is found such that $R'_\beta \equiv R'_\alpha$ — under this situation, we have the following two subcases:

*Case 1.1* ($\mathsf{s}'_\alpha = \mathsf{s}'_\beta$): Return $\langle \alpha, \beta, \overline{\upsilon}, \overline{\upsilon} \rangle$.

*Case 1.2* ($\mathsf{s}'_\alpha \neq \mathsf{s}'_\beta$): Return $\langle \alpha, \beta, \overline{\vartheta_{\alpha^f}}, \overline{\vartheta_{\beta^f}} \rangle$. The propagated vectors are computed by valuePropagation($\alpha$, $\overline{\vartheta_{\alpha^s}}$, $\beta$, $\overline{\vartheta_{\beta^s}}$).

*Case 2* ($R'_\alpha \Rightarrow R'_\beta$): Return $\langle \varpi_{\alpha^s}, \beta, \overline{\vartheta_{\varpi_{\alpha^s}}}, \overline{\vartheta_{\beta^f}} \rangle$. The null path $\varpi_{\alpha^s}$ originating (and terminating) in the start state of $\alpha$ is returned as $\alpha_m$ along with $\beta$ and the propagated vectors computed by valuePropagation invoked with the same parameters as above. Note that $\overline{\vartheta_{\varpi_{\alpha^s}}} = \overline{\vartheta_{\alpha^s}}$.

*Case 3* ($R'_\beta \Rightarrow R'_\alpha$): Return $\langle \alpha, \varpi_{q_{2,j}}, \overline{\vartheta_{\alpha^f}}, \overline{\vartheta_{\varpi_{q_{2,j}}}} \rangle$.

*Case 4* ($R'_\alpha \not\equiv R'_\beta$ *and* $R'_\alpha \not\Rightarrow R'_\beta$ *and* $R'_\beta \not\Rightarrow R'_\alpha$): Return $\langle \alpha, \Omega, \overline{\upsilon}, \overline{\upsilon} \rangle$. No path in $P_2$ can be found whose condition of execution is equal to or stronger/weaker than that of $\alpha$; then findEquivalentPath returns a non-existent path $\Omega$ in place of $\beta$. In this case, the other three values in the 4-tuple are not of any significance.

## 3.4 Correctness of symbolic value propagation as a method of equivalence checking

**Theorem 2** (Correctness of the approach). *An FSMD $M_1$ with no unreachable state*[3] *is contained in another FSMD $M_2$ ($M_1 \sqsubseteq M_2$), if for a finite path cover $P_1 = \{\alpha_1, \ldots, \alpha_{l_1}\}$ of $M_1$, there exists a path cover $P_2 = \{\beta_1, \ldots, \beta_{l_2}\}$ of $M_2$, such that*

[3]For taking care of useless paths, it suffices to check whether or not some cut-point is reachable from the reset state or not (using dfs or bfs).

**Algorithm 3** findEquivalentPath ($\alpha$, $\overline{\vartheta_{\alpha^s}}$, $q_{2,j}$, $\overline{\vartheta_{\beta^s}}$, $P_1$, $P_2$)

**Inputs:** A path $\alpha \in P_1$, the propagated vector at its start state $\overline{\vartheta_{\alpha^s}}$, a state $q_{2,j} \in M_2$ which is the C-corresponding or U-corresponding state of $\alpha^s$, the propagated vector $\overline{\vartheta_{\beta^s}}$ associated with $q_{2,j}$, and a path cover $P_1$ of $M_1$, a path cover $P_2$ of $M_2$.

**Outputs:** Let $\alpha_m = \alpha$ or $\varpi_{\alpha^s}$. An ordered tuple $\langle \alpha_m, \beta, \overline{\vartheta_{\alpha_m^f}}, \overline{\vartheta_{\beta^f}} \rangle$ such that $\alpha_m$ and $\beta$ are either U-equivalent or C-equivalent, and $\overline{\vartheta_{\alpha_m^f}}$ and $\overline{\vartheta_{\beta^f}}$ are the vectors that are to be propagated to $\alpha_m^f$ and $\beta^f$ respectively.

```
1: Let $R'_\alpha = R_\alpha(\Pi_2(\overline{\vartheta_{\alpha^s}})/\overline{v})$ and $s'_\alpha = s_\alpha(\Pi_2(\overline{\vartheta_{\alpha^s}})/\overline{v})$.
2: for each path $\beta \in P_2$ originating from $q_{2,j}$ do
3:     Let $R'_\beta = R_\beta(\Pi_2(\overline{\vartheta_{\beta^s}})/\overline{v})$ and $s'_\beta = s_\beta(\Pi_2(\overline{\vartheta_{\beta^s}})/\overline{v})$.
4:     if $R'_\alpha \equiv R'_\beta$ then
5:         if $s'_\alpha = s'_\beta$ then
6:             return $\langle \alpha, \beta, \overline{\upsilon}, \overline{\upsilon} \rangle$. $\boxed{\textbf{Case } 1.1}$
7:         else
8:             $\langle \overline{\vartheta_{\alpha^f}}, \overline{\vartheta_{\beta^f}} \rangle \leftarrow$ valuePropagation ($\alpha$, $\overline{\vartheta_{\alpha^s}}$, $\beta$, $\overline{\vartheta_{\beta^s}}$). $\boxed{\textbf{Case } 1.2}$
9:             return $\langle \alpha, \beta, \overline{\vartheta_{\alpha^f}}, \overline{\vartheta_{\beta^f}} \rangle$.
10:        end if
11:    else if $R'_\alpha \Rightarrow R'_\beta$ then
12:        $\langle \overline{\vartheta_{\varpi_{\alpha^s}}}, \overline{\vartheta_{\beta^f}} \rangle \leftarrow$ valuePropagation ($\varpi_{\alpha^s}$, $\overline{\vartheta_{\alpha^s}}$, $\beta$, $\overline{\vartheta_{\beta^s}}$). $\boxed{\textbf{Case } 2}$
13:        return $\langle \varpi_{\alpha^s}, \beta, \overline{\vartheta_{\varpi_{\alpha^s}}}, \overline{\vartheta_{\beta^f}} \rangle$.
14:    else if $R'_\beta \Rightarrow R'_\alpha$ then
15:        $\langle \overline{\vartheta_{\alpha^f}}, \overline{\vartheta_{\varpi_{q_{2,j}}}} \rangle \leftarrow$ valuePropagation ($\alpha$, $\overline{\vartheta_{\alpha^s}}$, $\varpi_{q_{2,j}}$, $\overline{\vartheta_{\beta^s}}$). $\boxed{\textbf{Case } 3}$
16:        return $\langle \alpha, \varpi_{q_{2,j}}, \overline{\vartheta_{\alpha^f}}, \overline{\vartheta_{\varpi_{q_{2,j}}}} \rangle$.
17:    end if
18: end for
19: return $\langle \alpha, \Omega, \overline{\upsilon}, \overline{\upsilon} \rangle$. $\boxed{\textbf{Case } 4}$
```

1. *each path of $P_1$ is either conditionally or unconditionally equivalent to some path of $P_2$ satisfying the correspondence relation of the respective start states and final states of the paths; thus, symbolically, $\forall i, 1 \leq i \leq l_1$,*

   1.1 *$\alpha_i \simeq \beta_j$ or $\alpha_i \simeq_c \beta_j$, and $\langle \alpha_i^s, \beta_j^s \rangle$ belongs to the set of corresponding (C-corresponding) state pairs, for some $j, 1 \leq j \leq l_2$, or*

   1.2 *$\alpha_i \simeq_c \varpi$, a null path of $M_2$, and $\langle \alpha_i^s, \varpi^s \rangle$ belongs to the set of corresponding (C-corresponding) state pairs; and*

2. *all paths of $P_1$ leading to the reset state will have unconditionally equivalent paths in $P_2$ leading to the reset state of $M_2$; thus, symbolically, $\forall \alpha_{k_1} \in P_1$ such that $\alpha_{k_1}^f$ is the reset state $q_{1,0}$ of $M_1$, $\exists \beta_{k_2} \in P_2$ such that $\beta_{k_2}^f$ is the reset state $q_{2,0}$ of $M_2$, and $\alpha_{k_1} \simeq \beta_{k_2}$.*

*Proof:* A path $\alpha_2$ is said to be consecutive to $\alpha_1$ if $\alpha_1^f = \alpha_2^s$. Since $P_1$ is a path cover of $M_1$, a computation $\mu_1$ of $M_1$ can be viewed as a concatenation of *consecutive* paths starting and ending at the reset state of $M_1$; symbolically, $\mu_1 = [\alpha_{i_1}, \alpha_{i_2}, \ldots, \alpha_{i_n}]$, where $\alpha_{i_j} \in P_1, 1 \leq j \leq n$, and $\alpha_{i_1}^s = \alpha_{i_n}^f = q_{1,0}$. From the hypotheses 1 and 2 of the theorem, it follows that there exists a sequence $S$ of paths $[\beta_{k_1}, \beta_{k_2}, \ldots, \beta_{k_n}]$, where $\beta_{k_j} \in P_2$ or is a null path of $M_2$, $1 \leq j < n$, and $\beta_{k_n} \in P_2$, such that $\alpha_{i_l} \simeq \beta_{k_l}$ or $\alpha_{i_l} \simeq_c \beta_{k_l}, 1 \leq l < n$ and $\alpha_{i_n} \simeq \beta_{k_n}$. For $S$ to represent a computation of $M_2$, it must be a concatenation of consecutive paths in $M_2$ starting from the reset state $q_{2,0}$ back to itself, which may not be the case because $\beta_{k_j}^f \neq \beta_{k_{j+1}}^s$ is possible when the path joining $\beta_{k_j}^f$ and $\beta_{k_{j+1}}^s$ is C-equivalent to a null path in $M_1$ (and hence is not present in $S$). Introduction of null paths at appropriate places does not alter the computation $\mu_1$ since null paths have conditions of execution *true* and null data transformations (by definition) and preserves consecutiveness property. Let $\mu_1'$ be a sequence obtained from $\mu_1$ by introducing such null paths at the appropriate places which have C-corresponding paths in $M_2$. The sequence yields a computation equivalent to $\mu_1$. The sequence $S'$ of paths of $M_2$ corresponding to $\mu_1'$ can be obtained by introducing at appropriate places in $S$ the paths of $M_2$ which are C-corresponding to the null paths of $M_1$ introduced in $\mu_1$ to obtain $\mu_1'$. This new sequence $S'$ now indeed represents a concatenated path that starts and ends at the reset state $q_{2,0}$. Furthermore, hypothesis 2 of the theorem implies that whatever mismatches might be present in the respective last but one paths in $\mu_1'$ and $S'$, they must get resolved when the respective last paths back to the reset states are traversed. Let $\mu_2$ be the computation of $M_2$ represented by sequence $S'$. Thus, the computations

$\mu_2$ and $\mu_1'$, and hence $\mu_1$, must be equivalent. ■

## 3.5 The overall verification method

To begin with the procedure of equivalence checking, we have to identify the cut-points in an FSMD followed by identification of paths and their corresponding characteristics involving the conditions and the data transformations. We also need to store the pairs of corresponding states in a structure, $\delta$ say, and the U-equivalent and the C-equivalent pairs of paths in $E_u$ and $E_c$, respectively.

The function containmentChecker (Algorithm 4) initializes all the above mentioned data structures, invokes correspondenceChecker (Algorithm 5) with the members of the set $\delta$ of the corresponding state pairs one by one to check whether for every path emanating from a state in the pair, there is a U-equivalent or C-equivalent path from the other member of the pair; depending on the result returned by correspondenceChecker, it outputs the decision whether the original FSMD is contained in the transformed FSMD or not. A list, called *LIST*, is maintained to keep track of the (candidate) C-equivalent pairs of paths visited along the chain of recursive invocations of correspondenceChecker invoked by containmentChecker. In case the equivalence checking procedure fails for a chain of paths, this *LIST* is output as a possible counterexample. Hence, note that every time correspondenceChecker is called from containmentChecker, the former's last parameter *LIST* is set to *empty*.

The verification procedure starts with the two reset states declared as corresponding states and also ends with the reset states as a corresponding state pair in case the two FSMDs are indeed equivalent. Otherwise, it terminates on encountering any of the following "failure" conditions: (i) given a path in one FSMD, it fails to discover its U-equivalent or C-equivalent path in the other one; (ii) it discovers that a propagated vector depicting some mismatches at a loop state is not a loop invariant and some of them do not get resolved in the loop. Note that while failure to find a U-equivalent path occurs in one step, failure to find a C-equivalent path may be detected only when the reset state is reached through several steps without finding a match in the variable values. A chain of C-equivalence may be obtained in the form $\alpha_1 \simeq_c \beta_1$ if $\alpha_2 \simeq_c$ if $\alpha_3 \simeq_c \beta_3, \ldots,$ if $\alpha_k \simeq \beta_k$, where $\alpha_k^f$ and $\beta_k^f$ are the reset states; when $\alpha_k \simeq \beta_k$ is identi-

```
Algorithm 4 containmentChecker ($M_1$, $M_2$)
Inputs: Two FSMDs $M_1$ and $M_2$.
Outputs: Whether $M_1$ is contained in $M_2$ or not, $P_1, P_2$: path covers of $M_1$ and $M_2$ respectively, $\delta$: the set of corresponding state pairs, $E_u$: ordered pairs $\langle \alpha, \beta \rangle$ of paths such that $\alpha \in P_1$ and $\beta \in P_2$, and $\alpha \simeq \beta$, $E_c$: ordered pairs $\langle \alpha, \beta \rangle$ of paths such that $\alpha \in P_1$ and $\beta \in P_2$, and $\alpha \simeq_c \beta$.
1: Incorporate cut-points in $M_1$ and $M_2$; Let $P_1$ ($P_2$) be the set of all paths of $M_1$ ($M_2$) from a cut-point to a cut-point having no intermediary cut-point, with each path having its condition of execution $R$, data transformation s, and output list $\theta$ computed.
2: Let $\delta$ be $\{\langle q_{1,0}, q_{2,0} \rangle\}$, and $E_u$ and $E_c$ be empty.
3: for each member $\langle q_{1,i}, q_{2,j} \rangle$ of $\delta$ do
4:     if correspondenceChecker ($q_{1,i}$, $q_{2,j}$, $P_1, P_2$, $\delta$, $E_u$, $E_c$, $LIST = \Phi$) returns "failure" then
5:         Report "May be $M_1 \not\sqsubseteq M_2$" and (exit).
6:     end if
7: end for
8: Report "$M_1 \sqsubseteq M_2$".
```

fied, the path pair $\langle \alpha_k, \beta_k \rangle$ is to be put in $E_u$ and the pairs $\langle \alpha_i, \beta_i \rangle, 1 \leq i \leq k-1$, in $E_c$ (after carrying out some further checks). Whereas, $\alpha_k \not\simeq \beta_k$ would result in a failure in the equivalence checking procedure, and the chain, kept track of by $LIST$, is output as a possible counterexample.

The central function is correspondenceChecker given in Algorithm 5. It takes as inputs two states $q_{1,i}$ and $q_{2,j}$ belonging to the FSMDs $M_1$ and $M_2$ respectively, and are in correspondence to each other; the path covers $P_1$ and $P_2$, the set of corresponding state pairs $\delta$, the set of U-equivalent path pairs $E_u$, the set of C-equivalent path pairs $E_c$, and the $LIST$. It returns "success" if for every path emanating from $q_{1,i}$, an equivalent path originating from $q_{2,j}$ is found; otherwise it returns "failure". The behaviour of the function is as follows. Dynamically, for any path $\alpha$ originating from the state $q_{1,i}$ of the original FSMD $M_1$, it invokes the function findEquivalentPath to find a U-equivalent or C-equivalent path $\beta$ originating from $q_{2,j}$ of the transformed FSMD $M_2$, where $\langle q_{1,i}, q_{2,j} \rangle$ is a corresponding or C-corresponding state pair. Recall that the function findEquivalentPath returns a 4-tuple $\langle \alpha_m, \beta, \overline{\vartheta_{\alpha_m^f}}, \overline{\vartheta_{\beta^f}} \rangle$ depending on the following cases: (i) if it fails to find a path $\beta$ such that $\alpha \simeq \beta$ or $\alpha \simeq_c \beta$, then it returns $\beta = \Omega$, where $\Omega$ represents a non-existent path, causing correspondenceChecker to return "failure" as shown in its step 5; (ii) if it finds a path $\beta$ such that $R_\alpha \equiv R_\beta$ or $R_\beta \Rightarrow R_\alpha$, then $\alpha_m$ is $\alpha$ itself; and (iii) if it finds a path $\beta$ such that $R_\alpha \Rightarrow R_\beta$ and

$R_\alpha \not\equiv R_\beta$, then $\alpha_m$ is returned as a null path from $q_{1,i}$. In the last two cases, the function correspondenceChecker next examines whether the propagated vectors $\overline{\vartheta_{\alpha_m^f}}$ and $\overline{\vartheta_{\beta^f}}$ computed after $\alpha_m$ and $\beta$ are equal or not.

$\overline{\vartheta_{\alpha_m^f}} \neq \overline{\vartheta_{\beta^f}}$: Unequal propagated vectors imply that U-equivalence could not be established, and hence further symbolic value propagation is required. However, the following checks are carried out first.

*Loop-invariance*: Whether a loop has been crossed over is checked in step 7, and if so, a check for loop invariance of the propagated vectors $\overline{\vartheta_{\alpha_m^f}}$ and $\overline{\vartheta_{\beta^f}}$ is carried out with the aid of the function loopInvariant. In case a failure in loop invariance is detected for either of the propagated vectors, correspondenceChecker returns "failure".

*Extendability*: Next, checks are made to ensure that neither of the end states $\alpha_m^f$ and $\beta^f$ is a reset state. Since a computation does not extend beyond the reset state, reaching a reset state with C-equivalence (and not U-equivalence) results in returning failure by the correspondenceChecker as shown in step 10.

If $\overline{\vartheta_{\alpha_m^f}} \neq \overline{\vartheta_{\beta^f}}$ and both the checks for loop invariance and end states being a reset state resolve in success, then $\langle \alpha_m, \beta \rangle$ is appended to *LIST*, and correspondenceChecker calls itself recursively with the arguments $\alpha_m^f$ and $\beta^f$ (step 14) to continue searching for equivalent paths.

$\overline{\vartheta_{\alpha_m^f}} = \overline{\vartheta_{\beta^f}}$: Attainment of equal propagated vectors signify discovery of U-equivalent paths. Consequently, the data structures $\delta$ and $E_u$ get updated. Notice that these steps are executed only after the recursive calls to correspondenceChecker terminate, i.e., a U-equivalent pair has been found. It is to be noted that a state pair qualifies to be a member of $\delta$ if the propagated vectors computed for these states match totally, i.e., when $\overline{\vartheta_{\alpha_m^f}} = \overline{\vartheta_{\beta^f}}$.

When the control reaches step 22 of correspondenceChecker, it implies that for every chain of paths that emanates from the state $q_{1,i}$, there exists a corresponding chain of paths emanating from $q_{2,j}$ such that their final paths are U-equivalent. Note that $q_{1,i}$ and $q_{2,j}$ are the respective final states of the last member of *LIST*. Hence, the last member of *LIST* gets added to the set $E_c$ in accordance with Definition 7 and is removed from *LIST*. The remaining (preceding) members are yet to be declared C-equivalent because all the paths emanating from the final state of the last path of the

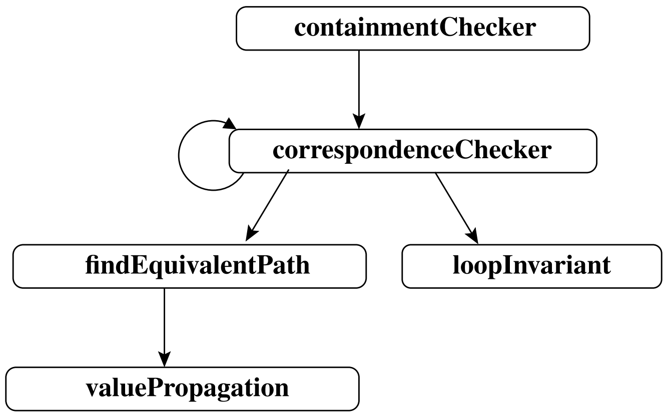


Figure 3.5: Call graph of the proposed verification method.

updated *LIST* have not yet been examined. The members of *LIST* are also displayed as a part of the report generated whenever one of the “failure” conditions mentioned above is encountered to aid in the process of debugging.

A call graph of the proposed verification method is given in Figure 3.5. The correctness and the complexity of the method have been treated in Section 3.6.

### 3.5.1 An illustrative example

In this section, the working of our verification procedure is explained with the example given in Figure 3.6. Figure 3.6(a) and Figure 3.6(b) show two FSMDs before and after scheduling. The definition of $a$ in Figure 3.6(a) occurs before the loop ($q_{1,1} \rightarrow q_{1,2} \rightarrow q_{1,1}$), whereas in Figure 3.6(b), it has been moved after the loop ($q_{2,1} \rightarrow q_{2,2} \rightarrow q_{2,1}$) – an instance of code motion across loop. Such code motions reduce register life times (as in this case) and may also minimize the overall register usage. Moreover, the definition of $z$ has been moved from both the (*true* and *false*) branches to the predecessor block which is an instance of boosting up – a uniform code motion technique, and the definition of $g$ has been moved from one of the branches to the predecessor block which is an instance of speculation – a non-uniform code motion technique.

**Example 5.** The example given in Figure 3.6(a) represents the original behaviour $M_1$ and Figure 3.6(b) represents its transformed behaviour $M_2$. The following steps summarize the progress of the verification method. To begin with, $q_{1,0}$ and $q_{2,0}$ are

**Algorithm 5** correspondenceChecker ($q_{1,i}$, $q_{2,j}$, $P_1$, $P_2$, $\delta$, $E_u$, $E_c$, $LIST$)

**Inputs:** Two states $q_{1,i} \in M_1$ and $q_{2,j} \in M_2$, two path covers $P_1$ of $M_1$ and $P_2$ of $M_2$, $\delta$: the set of corresponding state pairs, $E_u$ and $E_c$ for storing the pairs of U-equivalent and C-equivalent paths, respectively, and $LIST$: a list of paths maintained to keep track of candidate C-equivalent paths.

**Outputs:** Returns "success" if for every path emanating from $q_{1,i}$ there is an equivalent path originating from $q_{2,j}$ in $M_2$, otherwise returns "failure". Also updates $\delta$, $E_u$, $E_c$ and $LIST$.

```
1:  for each path $\alpha \in P_1$ emanating from $q_{1,i}$ do
2:    if $\Pi_1(\overline{\vartheta_{0,i}}) \wedge R_\alpha(\overline{v})\{\Pi_2(\overline{\vartheta_{0,i}})/\overline{v}\} \not\equiv false$ then
3:      $\langle \alpha_m, \beta, \overline{\vartheta'_{\alpha_m^f}}, \overline{\vartheta'_{\beta^f}} \rangle \leftarrow$ findEquivalentPath ($\alpha$, $\overline{\vartheta_{0,i}}$, $q_{2,j}$, $\overline{\vartheta_{1,j}}$, $P_1$, $P_2$).
4:      if $\beta = \Omega$ then
5:        Report "equivalent path of $\alpha$ may not be present in $M_2$,"
          display $LIST$ and return (failure).
6:      else if $\overline{\vartheta_{\alpha_m^f}} \neq \overline{\vartheta_{\beta^f}}$ /* satisfied by cases 1.2, 2, 3 of Algorithm 3 */ then
7:        if ($\alpha_m^f$ appears as the final state of some path already in $LIST$ $\wedge$
            !loopInvariant ($\alpha_m$, $\overline{\vartheta'_{\alpha_m^f}}$, $\overline{\vartheta_{\alpha_m^f}}$, $\overline{\vartheta'_{\beta^f}}$)) $\vee$
           ($\beta^f$ appears as the final state of some path already in $LIST$ $\wedge$
            !loopInvariant ($\beta$, $\overline{\vartheta'_{\beta^f}}$, $\overline{\vartheta_{\beta^f}}$, $\overline{\vartheta'_{\alpha_m^f}}$)) then
8:          Report "propagated values are not loop invariant,"
            display $LIST$ and return (failure).
9:        else if $\alpha_m^f = q_{1,0}$ $\vee$ $\beta^f = q_{2,0}$ then
10:         Report "reset states reached with unresolved mismatch,"
            display $LIST$ and return (failure).
11:       else
12:         $\overline{\vartheta_{\alpha_m^f}} \leftarrow \overline{\vartheta'_{\alpha_m^f}}$; $\overline{\vartheta_{\beta^f}} \leftarrow \overline{\vartheta'_{\beta^f}}$. /* candidate C-equivalence */
13:         Append $\langle \alpha_m, \beta \rangle$ to $LIST$.
14:         return correspondenceChecker ($\alpha_m^f$, $\beta^f$, $P_1$, $P_2$, $\delta$, $E_u$, $E_c$, $LIST$).
15:       end if /* loopInvariant */
16:     else
17:       $E_u \leftarrow E_u \bigcup \{\langle \alpha_m, \beta \rangle\}$. /* U-equivalence */
18:       $\delta \leftarrow \delta \bigcup \{\langle \alpha_m^f, \beta_m^f \rangle\}$. /* $\alpha_m = \alpha$ in this case */
19:     end if /* $\beta = \Omega$ */
20:   end if
21: end for
22: $E_c \leftarrow E_c \bigcup$ {last member of $LIST$}.
23: $LIST \leftarrow LIST -$ {last member of $LIST$}.
24: return (success).
```

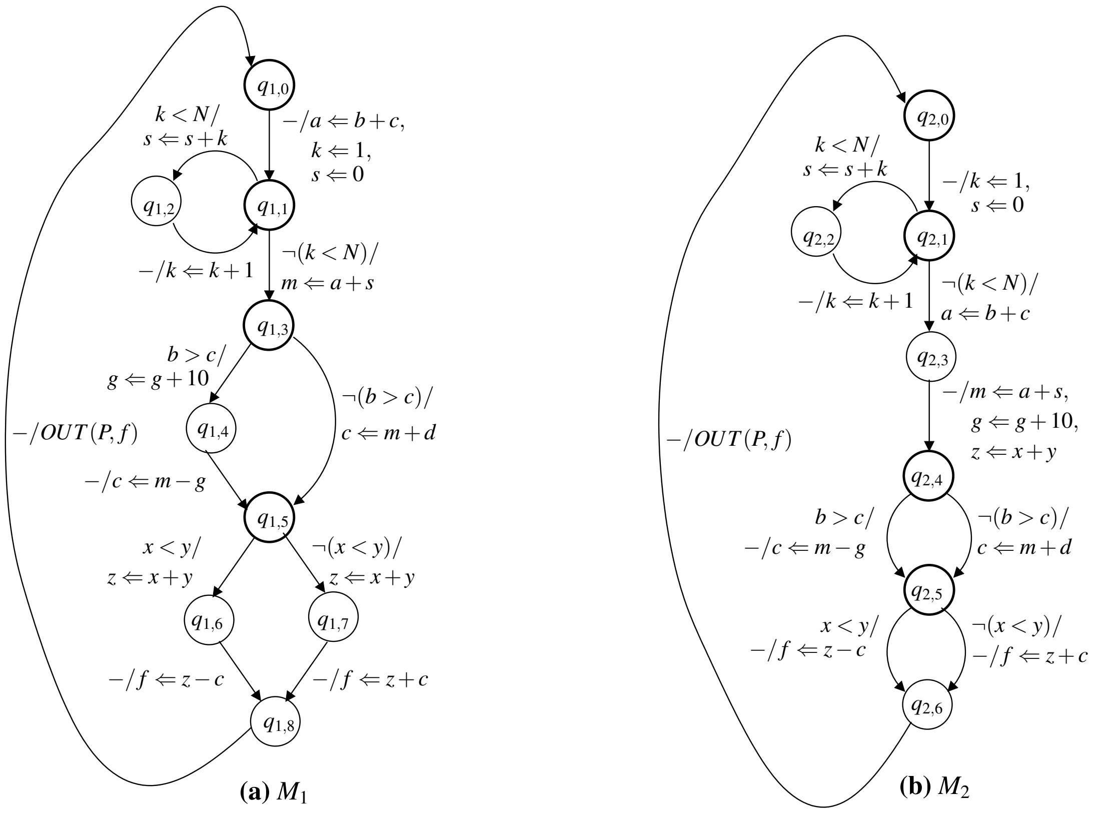


Figure 3.6: FSMDs before and after scheduling.

considered to be corresponding states.

1) The path characteristics of $q_{1,0} \twoheadrightarrow q_{1,1}$ and $q_{2,0} \twoheadrightarrow q_{2,1}$ are found to match in all aspects other than the definition of $a$. Hence, they are declared to be candidate C-equivalent and the path pair is stored in the *LIST* (step 13 of correspondenceChecker).

2) Next, the loops $q_{1,1} \twoheadrightarrow q_{1,1}$ and $q_{2,1} \twoheadrightarrow q_{2,1}$ are compared. The variables $s$ and $k$ are found to be updated identically in both the loops while the values for $a$, viz., $b+c$ in $M_1$ and the symbolic value "$a$" in $M_2$, along with those of $b$ and $c$ remain unmodified in the loops; hence, it is concluded that code motion across loop may have taken place and the paths representing the loops are appended to the *LIST*.

3) The paths $q_{1,1} \twoheadrightarrow q_{1,3}$ and $q_{2,1} \twoheadrightarrow q_{2,4}$ are analyzed next. The definition of $a$ in the latter path is found to be equivalent to that in $M_1$. The values of $g$ and $z$, however, mismatch in the paths being compared and consequently, the paths are also put in the *LIST*.

4) When the paths $q_{1,3} \overset{b>c}{\twoheadrightarrow} q_{1,5}$ and $q_{2,4} \overset{b>c}{\twoheadrightarrow} q_{2,5}$ are compared, the values of the variables $g$ and $c$ match but the mismatch for $z$ still persists; therefore, these paths are put in the *LIST* as well.

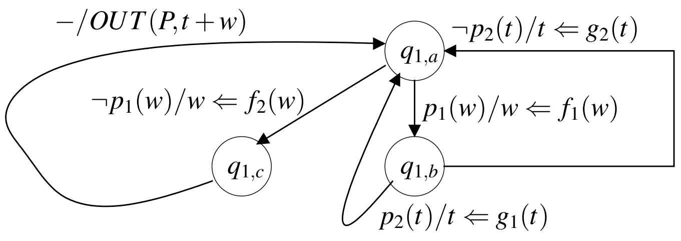
5) Finally, on comparing the paths $q_{1,5} \overset{x<y}{\twoheadrightarrow} q_{1,0}$ and $q_{2,5} \overset{x<y}{\twoheadrightarrow} q_{2,0}$, all values are

found to match. Consequently, these pair of paths are declared to be U-equivalent. Note that as per Definition 7, a pair of candidate C-equivalent pair of paths can be declared to be actually C-equivalent when all the paths emanating from that pair are found to be U-equivalent or C-equivalent. Hence, the paths $q_{1,3} \xrightarrow{b>c} q_{1,5}$ and $q_{2,4} \xrightarrow{b>c} q_{2,5}$ cannot be declared as C-equivalent yet.
6) Owing to the depth-first traversal of the FSMDs (achieved through recursion in step 14 of correspondenceChecker), the paths $q_{1,5} \xrightarrow{\neg(x<y)} q_{1,0}$ and $q_{2,5} \xrightarrow{\neg(x<y)} q_{2,0}$ are examined next. Note that the *LIST* available in this step is the same as that of step 5. Again these paths are found to be U-equivalent and now the paths $q_{1,3} \xrightarrow{b>c} q_{1,5}$ and $q_{2,4} \xrightarrow{b>c} q_{2,5}$ are declared to be C-equivalent.
7) Now the paths $q_{1,3} \xrightarrow{\neg(b>c)} q_{1,5}$ and $q_{2,4} \xrightarrow{\neg(b>c)} q_{2,5}$ are compared. It is found that they differ in the values of the variables $g$ and $z$; however, $g$ is no longer a live variable at $q_{1,5}$ and $q_{2,5}$ – hence its value can be ignored.
8-9) With the propagated value of $z$, the pairs of paths $q_{1,5} \xrightarrow{x<y} q_{1,0}$ and $q_{2,5} \xrightarrow{x<y} q_{2,0}$, and $q_{1,5} \xrightarrow{\neg(x<y)} q_{1,0}$ and $q_{2,5} \xrightarrow{\neg(x<y)} q_{2,0}$ are declared to be U-equivalent and all the path pairs present in the LIST are declared to be C-equivalent. ■

### 3.5.2 An example of dynamic loop scheduling

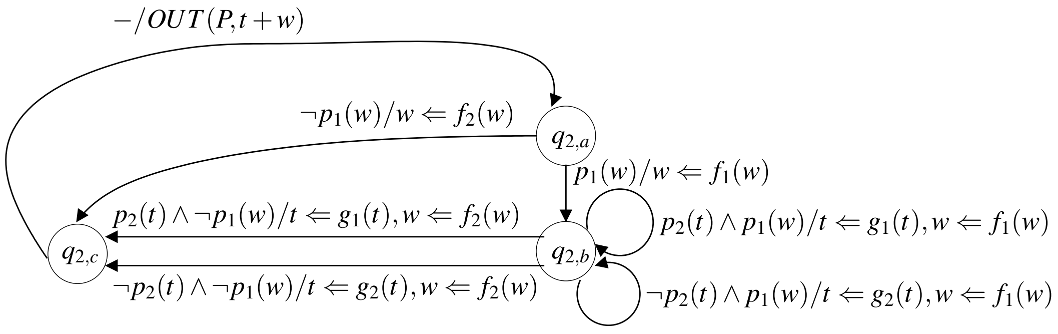
The dynamic loop scheduling (DLS) [135] transformation modifies the control structure of a behaviour by introducing new branches while keeping all the loop feedback edges intact. As already mentioned in Section 3.3.2, the *LIST* has been introduced to detect crossing over of loops. This is a deviation from [21] where back edges were used for detection of loops. The new method of loop detection aids in verifying DLS transformations [135] which [21] cannot handle. The example in Figure 3.7 is used to illustrate verification of DLS transformations.

**Example 6.** The example given in Figure 3.7(a) shows the original behaviour and Figure 3.7(b) shows its transformed version after application of DLS. Note that the states $q_{1,a}$, $q_{1,b}$ of FSMD $M_1$ and the states $q_{2,a}$, $q_{2,b}$ of FSMD $M_2$ qualify as cut-points. Initially, $q_{1,a}$ and $q_{2,a}$ are considered as corresponding states and the verification procedure then proceeds in the following sequence.
1) The path pair $\langle q_{1,a} \twoheadrightarrow q_{1,b}, q_{2,a} \twoheadrightarrow q_{2,b} \rangle$ is declared as U-equivalent.
2) The path pair $\langle q_{1,a} \xrightarrow{\neg p_1} q_{1,a}, q_{2,a} \xrightarrow{\neg p_1} q_{2,a} \rangle$ is declared as U-equivalent.



$-/OUT(P,t+w)$
$q_{1,a}$
$\neg p_2(t)/t \Leftarrow g_2(t)$
$\neg p_1(w)/w \Leftarrow f_2(w)$
$p_1(w)/w \Leftarrow f_1(w)$
$q_{1,c}$
$q_{1,b}$
$p_2(t)/t \Leftarrow g_1(t)$

(a) Original behaviour $M_1$



$-/OUT(P,t+w)$
$\neg p_1(w)/w \Leftarrow f_2(w)$
$q_{2,a}$
$p_1(w)/w \Leftarrow f_1(w)$
$p_2(t) \wedge \neg p_1(w)/t \Leftarrow g_1(t), w \Leftarrow f_2(w)$
$p_2(t) \wedge p_1(w)/t \Leftarrow g_1(t), w \Leftarrow f_1(w)$
$q_{2,c}$
$q_{2,b}$
$\neg p_2(t) \wedge \neg p_1(w)/t \Leftarrow g_2(t), w \Leftarrow f_2(w)$
$\neg p_2(t) \wedge p_1(w)/t \Leftarrow g_2(t), w \Leftarrow f_1(w)$

(b) After DLS $M_2$

Figure 3.7: An example of dynamic loop scheduling (DLS).

Next, we consider the paths emanating from the corresponding states $q_{1,b}$ and $q_{2,b}$.
3) For the path $q_{1,b} \xrightarrow{p_2} q_{1,a}$ in $M_1$, it is found that there are two paths in $M_2$, namely $q_{2,b} \xrightarrow{p_2 \wedge p_1} q_{2,b}$ and $q_{2,b} \xrightarrow{p_2 \wedge \neg p_1} q_{2,a}$, emanating from $q_{2,b}$ in $M_2$ whose conditions of execution are stronger than the path of $M_1$. This results in declaring the paths $q_{1,b} \xrightarrow{p_2} q_{1,a}$ and $\varpi_{q_{2,b}}$ as candidate C-equivalent and the path pair is stored in the *LIST*.
4) The paths $q_{1,a} \twoheadrightarrow q_{1,b}$ (with respect to the propagated vector $\langle p_2(t), \langle \mathbf{g_1}(\mathbf{t}), w\rangle\rangle$) and $q_{2,b} \xrightarrow{p_2 \wedge p_1} q_{2,b}$ (w.r.t. $\langle \top, \langle \mathbf{t}, w\rangle\rangle$) are now found to be U-equivalent since the condition of execution and the data transformation of the former path match with those of the latter. It is to be noted that after considering the latter path in $M_2$, one ends up in $q_{2,b}$ which is the final state of a path (specifically, $\varpi_{q_{2,b}}$) that is already present in the *LIST*. However, since U-equivalence was established by the time this state is (re)visited, the equivalence of the paths can be ascertained, and the call to loopInvariant is no longer required.
5) First of all, note that when this step is executed, the invoked call to this instance of correspondenceChecker has $\{\langle q_{1,b} \xrightarrow{p_2} q_{1,a}, \varpi_{q_{2,b}}\rangle\}$ in the *LIST* as well. A similar U-equivalence is now established between the paths $q_{1,a} \xrightarrow{\neg p_1} q_{1,a}$ of $M_1$ and $q_{2,b} \xrightarrow{p_2 \wedge \neg p_1} q_{2,a}$ of $M_2$.

Since eventually for all paths emanating from the final state of $q_{1,b} \xrightarrow{p_2}\!\!\!\!\!\!\rightarrow q_{1,a}$, a U-equivalent path is discovered that originates from the final state of $\varpi_{q_{2,b}}$, the path pair $\langle q_{1,b} \xrightarrow{p_2}\!\!\!\!\!\!\rightarrow q_{1,a}, \varpi_{q_{2,b}}\rangle$ is declared as *actually* C-equivalent, i.e. put in $E_c$ and the *LIST* is rendered empty by removing its last (and only) entry. Note that unless both the paths from $q_{1,a}$, i.e. $q_{1,a} \xrightarrow{p_1}\!\!\!\!\!\!\rightarrow q_{1,b}$ and $q_{1,a} \xrightarrow{\neg p_1}\!\!\!\!\!\!\rightarrow q_{1,a}$ are treated, *LIST* is not updated – as borne out by the fact that *LIST* update takes place after the loop 1 - 21 in correspondenceChecker.
6) The paths $q_{1,b} \xrightarrow{\neg p_2}\!\!\!\!\!\!\rightarrow q_{1,a}$ and $\varpi_{q_{2,b}}$ are considered as candidate C-equivalent, similar to step 3.
7-8) The path pairs $\langle q_{1,a} \twoheadrightarrow q_{1,b}, q_{2,b} \xrightarrow{\neg p_2 \wedge p_1}\!\!\!\!\!\!\rightarrow q_{2,b}\rangle$ and $\langle q_{1,a} \xrightarrow{\neg p_1}\!\!\!\!\!\!\rightarrow q_{1,a}, q_{2,b} \xrightarrow{\neg p_2 \wedge \neg p_1}\!\!\!\!\!\!\rightarrow q_{2,a}\rangle$ are declared as U-equivalent, and $\langle q_{1,b} \xrightarrow{\neg p_2}\!\!\!\!\!\!\rightarrow q_{1,a}, \varpi_{q_{2,b}}\rangle$ is declared as actually C-equivalent. ■

Now let us consider what would have gone wrong if back edges were used to detect loops instead of *LIST*. The first two steps of the method described in [21] would have been identical to the ones given above, and the states $q_{1,b}$ and $q_{2,b}$ would have been declared as corresponding states. In step 3, the method in [21] would find the paths $q_{1,b} \xrightarrow{p_2}\!\!\!\!\!\!\rightarrow q_{1,a}$ and $\varpi_{q_{2,b}}$ as candidate C-equivalent (similar to ours); however, it would also detect $q_{1,b} \xrightarrow{p_2}\!\!\!\!\!\!\rightarrow q_{1,a}$ as a back edge to $q_{1,a}$. While entering the loop at $q_{1,a}$, the variable $t$ had the symbolic value "$t$", but after step 3 the propagated vector computed for $q_{1,a}$ would have the value $g_1(t)$ for $t$. Clearly, there is a mismatch in the values for $t$, and therefore the method in [21] would terminate declaring the two FSMDs to be possibly non-equivalent. It is important to note that in case of code motions across loops, some paths will be detected to be C-equivalent while entering the loops. Hence, whenever the entry/exit state is revisited without resolving the mismatches, the state will definitely appear as a final state of some path in *LIST*, thereby leading to detection of loops (and without having any explicit knowledge about back edges).

The bisimulation based method in [115] can handle the control structure modifications stated in the path based scheduling method [33]. In general, during path based scheduling, two consecutive paths get concatenated into one whose condition of execution is obtained by taking the conjunction of those of its constituent paths. However, unlike [33], DLS may introduce new branches by concatenating paths that lie within a loop with the "exit" path of that loop (such as the branches $q_{2,b} \xrightarrow{p_2 \wedge \neg p_1} q_{2,c}$ and $q_{2,b} \xrightarrow{\neg p_2 \wedge \neg p_1} q_{2,c}$ in Figure 3.7(b)). A primary requirement for the equivalence checking method of [115] is that the loop exit conditions must be identical in order to find whether two states are in a relation (a concept similar to our corresponding states); consequently, the method of [115] results in a *failure* on application of DLS. For a more sophisticated example of DLS, one may refer to [91], which both the path extension method [91, 95] and the symbolic value propagation method described in this work can handle.

## 3.6 Correctness and complexity of the equivalence checking procedure

### 3.6.1 Correctness

**Theorem 3** (Partial correctness)**.** *If the verification method terminates in step 8 of the function containmentChecker, then $M_1 \sqsubseteq M_2$.*

*Proof:* The proof is tantamount to ascertaining that if the verification method terminates in step 8 of containmentChecker, then both hypotheses 1 and 2 of Theorem 7 are satisfied by the path covers $P_1$ and $P_2$ for the FSMDs $M_1$ and $M_2$, respectively. The path covers $P_1$ and $P_2$ comprise of paths starting from and ending in cut-point(s) without having any intermediary cut-point. The set $E_u$ of U-equivalent paths and $E_c$ of C-equivalent paths are updated in steps 17 and 22, respectively, of correspondenceChecker. The fact that the pair of paths added to the set $E_u$ and $E_c$ are indeed U-equivalent and C-equivalent, respectively, is affirmed by the function findEquivalentPath.

Now, we need to prove that $E = E_u \bigcup E_c$ contains a member for each path in $P_1$. Suppose a path $\alpha$ exists in $P_1$ that does not have a corresponding member in $E$. Absence of $\alpha$ in $E$ indicates that the path has not been considered at all during execution of the verification method. Since the state $\alpha^s$ is reachable (by definition) there must be some other path $\alpha'$ that leads to it. Let us consider the following cases:

$\alpha' \in E$ : Then $E$ must contain some member of the form $\langle \alpha', \beta' \rangle$, where $\beta' \in P_2$ and either (i) $\alpha' \simeq_c \beta'$ which means $\alpha$ would definitely have been considered in some subsequent recursive call of correspondenceChecker, or (ii) $\alpha' \simeq \beta'$ which means the end state of $\alpha'$, i.e., the start state of $\alpha$, must be a member of $\delta$ and $\alpha$ must have eventually been considered as given in step 4 of containmentChecker. (contradiction)

$\alpha' \notin E$ : In such a case, one should consider the path $\alpha''$ that leads to $\alpha'$. Now, these two cases hold for $\alpha''$ as well. A repetitive application of the argument lands up in the paths emanating from the reset state $q_{1,0}$, which is a member of $\delta$ by step 2 of containmentChecker; here steps 3 and 4 of containmentChecker ensure invocation of correspondenceChecker with $q_{1,i} = q_{1,0}$ and step 1 of correspondenceChecker ensures that paths from $q_{1,0}$ must have been treated, thereby again leading to a contradiction.

What remains to be proved is that hypothesis 2 of Theorem 7 holds when the verification method terminates in step 8 of containmentChecker. Suppose it does not. Then there exist paths $\alpha \in P_1$ and $\beta \in P_2$ such that $\alpha^f = q_{1,0}$, $\beta^f = q_{2,0}$ and $\alpha \simeq_c \beta$. It would result in trying to extend a path beyond the reset states, and thus, the function correspondenceChecker returns failure to containmentChecker as shown in step 10, whereupon the latter terminates in step 5 and not in step 8. (contradiction) ■

**Theorem 4** (Termination)**.** *The verification method always terminates.*

*Proof:* With respect to the call graph of Figure 3.5, we prove the termination of the modules in a bottom-up manner. The functions valuePropagation and loopInvariant obviously execute in finite time since the former involves comparison of two path characteristics and two propagated vectors, whereas the latter involves comparison of two propagated vectors only. The *for*-loop in findEquivalentPath is executed only $\|P_2\|$ number of times which is finite. The outermost *for*-loop in the function correspondenceChecker can be executed $\|P_1\|$ times which is also finite. Whenever correspondenceChecker is invoked with the states $q_{1,i}$ and $q_{2,j}$ as arguments, there is a possibility that the function is recursively invoked with the end states of some path $\alpha$ originating from $q_{1,i}$ and some path $\beta$ originating from $q_{2,j}$. Now, it remains to be shown that correspondenceChecker can invoke itself recursively only finite number of times. Let us associate the tuple $\langle n_0, n_1 \rangle \in \mathbb{N}^2$ to each invocation of correspondenceChecker($\alpha_m^f, \beta_m^f, \ldots$) (step 14), where $n_0$ and $n_1$ denote the number of paths that lie ahead of $\alpha_m^f$ and $\beta_m^f$ in FSMD $M_1$ and FSMD $M_2$, respectively. (Note that one cannot go beyond the reset states.) Also, let $\langle n'_0, n'_1 \rangle < \langle n_0, n_1 \rangle$ if $n'_0 < n_0$ or, $n'_0 = n_0$ and $n'_1 < n_1$. Note that in the 4-tuple returned by findEquivalentPath in step 3 of correspondenceChecker, $\alpha_m$ and $\beta$ both cannot be null paths simultaneously. Hence, a sequence of recursive invocations of correspondenceChecker can be represented by a sequence of these tuples $\langle n_0, n_1 \rangle > \langle n'_0, n'_1 \rangle > \langle n''_0, n''_1 \rangle$, and so on; specifically, this is a strictly decreasing sequence. Since $\mathbb{N}^2$ is a well-founded set [118] of pairs of nonnegative numbers having no infinite decreasing sequence, correspondenceChecker cannot call itself recursively infinite times. The only loop in containmentChecker depends upon the size of $\delta$, the set of corresponding states. Note that correspondenceChecker is called in this loop for every member of $\delta$ only once. Since the number of states in both the FSMDs is finite, the number of elements in $\delta$ has to be finite. ■

### 3.6.2 Complexity

The complexity of the overall verification method is in the order of product of the following two terms: (i) the complexity of finding a U-equivalent or a C-equivalent path for a given path from a state, and (ii) the number of times we need to find such a path. The first term is the same as the complexity of findEquivalentPath($\alpha$,$q_{2,j}$,$\cdots$) which tries to find a path $\beta$ starting from $q_{2,j} \in M_2$ such that $\beta \simeq \alpha$ or $\beta \simeq_c \alpha$. Let the number of states in $M_2$ be $n$ and the maximum number of parallel edges between any two states be $k$. Therefore, the maximum possible state transitions from a state are $k \cdot n$. The condition of execution associated with each transition emanating from a state is distinct. The function checks all transitions from $q_{2,j}$ in the worst case. Note that the conditions of execution and the data transformations of the paths are stored as normalized formulae [141] by the function containmentChecker. If $\|F\|$ be the length of the normalized formula (in terms of the number of variables along with that of the operations in $F$), then the complexity of normalization of $F$ is $O(2^{\|F\|})$ due to multiplication of normalized sums. As such, all the paths in the FSMDs will have their data transformations and conditions of execution computed during the initialization steps in the function containmentChecker. However, the function findEquivalentPath needs to substitute the propagated values in these entities necessitating multiplication. Hence, the complexity of finding a U-equivalent or a C-equivalent path is $O(k \cdot n \cdot 2^{\|F\|})$. On finding a C-equivalent path, symbolic value propagation is carried out in $O(2^{\|F\|} \cdot |V_1 \bigcup V_2|)$ time. So, the overall complexity is $O(2^{\|F\|} \cdot (k \cdot n + |V_1 \bigcup V_2|))$.

The second term is given by the product of (i) the number of times correspondenceChecker is called from containmentChecker, which is the same as the size of the set of corresponding states $\delta$, and (ii) the number of recursive calls made to correspondenceChecker (in the worst case). We notice that if the number of states for the original FSMD is $n$, then the number of states for the transformed FSMD is in $O(n)$ for both path based scheduler [33] and SPARK [64]. So, for simplicity, let the number of states in $M_2$ be $n$ as well. In the worst case, all the states of $M_1$ may be cut-points and the number of paths in $M_1$ is at most $k \cdot n \cdot (n-1)/2$. In this case the correspondenceChecker can recursively call itself as many as $(n-1)$ times leading to consideration of $k \cdot (n-1) + k^2 \cdot (n-1) \cdot (n-2) + \cdots + k^{(n-1)} \cdot (n-1) \cdot (n-2) \cdots 2 \cdot 1 \simeq k^{(n-1)} \cdot (n-1)^{(n-1)}$ number of paths. Also, $\|\delta\| \leq n$. Therefore, the complexity of the overall method is $O((k \cdot n + |V_1 \bigcup V_2|) \cdot 2^{\|F\|} \cdot n \cdot k^{(n-1)} \cdot (n-1)^{(n-1)})$ in the worst case. It is important to note that in [90], the authors had neglected the time complexity of computing the path characteristics of the concatenated paths that result from path extensions. Upon considering the same, the worst case time complexity of the presented method is found to be identical to that of [90].

## 3.7 Experimental Results

Table 3.1: Verification results based on our set of benchmarks

| Benchmarks | Original FSMD | | Transformed FSMD | | #Variable | | #across | Maximum | Time (ms) | |
|---|---|---|---|---|---|---|---|---|---|---|
| | #state | #path | #state | #path | com | uncom | loops | mismatch | PE | SVP |
| BARCODE-1 | 33 | 54 | 25 | 56 | 17 | 0 | 0 | 3 | 20.1 | 16.2 |
| DCT-1 | 16 | 1 | 8 | 1 | 41 | 6 | 0 | 6 | 6.3 | 3.6 |
| DIFFEQ-1 | 15 | 3 | 9 | 3 | 19 | 3 | 0 | 4 | 5.0 | 2.6 |
| EWF-1 | 34 | 1 | 26 | 1 | 40 | 1 | 0 | 1 | 4.2 | 3.6 |
| LCM-1 | 8 | 11 | 4 | 8 | 7 | 2 | 1 | 4 | × | 2.5 |
| IEEE754-1 | 55 | 59 | 44 | 50 | 32 | 3 | 4 | 3 | × | 17.7 |
| LRU-1 | 33 | 39 | 32 | 38 | 19 | 0 | 2 | 2 | × | 4.0 |
| MODN-1 | 8 | 9 | 8 | 9 | 10 | 2 | 0 | 3 | 5.6 | 2.5 |
| PERFECT-1 | 6 | 7 | 4 | 6 | 8 | 2 | 2 | 2 | × | 0.9 |
| QRS-1 | 53 | 35 | 24 | 35 | 25 | 15 | 3 | 19 | × | 15.9 |
| TLC-1 | 13 | 20 | 7 | 16 | 13 | 1 | 0 | 2 | 9.1 | 4.1 |

The verification procedure presented in this chapter has been implemented in $C$ on a 2.0 GHz Intel® Core™2 Duo machine and satisfactorily tested on several benchmarks. The results are provided in Table 3.1. Some of these benchmarks, such as BARCODE, LCM, QRS and TLC, are control intensive; some are data intensive, such as DCT, DIFFEQ and EWF, whereas some are both control and data intensive, such as IEEE754 and LRU. The transformed FSMD is obtained from the original one in multi-

Table 3.2: Verification results based on the benchmarks presented in [95]

| Benchmarks | PE (in ms) | | | SVP (in ms) | | |
|---|---|---|---|---|---|---|
| | BB-based | path based | SPARK | BB-based | path based | SPARK |
| BARCODE-2 | 12.1 | 15.4 | 19.8 | 10.1 | 13.1 | 12.0 |
| DCT-2 | 3.3 | 3.1 | 3.3 | 2.3 | 2.6 | 2.7 |
| DHRC-2 | 29.6 | 28.1 | 29.4 | 25.9 | 28.0 | 26.9 |
| DIFFEQ-2 | 1.4 | 2.5 | 3.4 | 1.2 | 1.4 | 1.4 |
| EWF-2 | 2.2 | 1.5 | 2.0 | 1.4 | 1.3 | 1.8 |
| GCD-2 | 3.0 | 4.1 | 3.0 | 1.6 | 1.7 | 1.3 |
| IEEE754-2 | 18.3 | 14.4 | 25.8 | 20.1 | 14.6 | 20.1 |
| LRU-2 | 5.2 | 4.8 | 7.6 | 4.3 | 4.0 | 4.1 |
| MODN-2 | 2.4 | 2.4 | 5.4 | 1.5 | 1.5 | 2.3 |
| PERFECT-2 | 1.9 | 1.7 | 2.4 | 0.8 | 0.9 | 1.2 |
| PRAWN-2 | 192.8 | 223.6 | 217.8 | 52.2 | 59.5 | 48.9 |
| TLC-2 | 3.4 | 6.9 | 2.9 | 3.0 | 3.3 | 2.9 |

ple steps. First, we feed the original FSMD to the synthesis tool SPARK [64] to get an intermediate FSMD which is then converted into the (final) transformed FSMD manually according to a path-based scheduler [33] and accounting for induction variable elimination. This multiple-step process helps us ascertain that our method can perform equivalence checking successfully when both control structure has been modified and code motions have occurred. It is to be noted that we prevent SPARK from applying loop shifting and loop unrolling transformations since they cannot be handled by our verifier presently. The column "#across loops" in Table 3.1 represents the number of code motions across loops; the '0' entries indicate that the transformed FSMDs contain no operation that has been moved across loop(s). However, the transformations do help in reducing the number of states (BARCODE, DCT, EWF, etc.) and/or the number of paths (TLC). The column Maximum Mismatch displays the maximum number of mismatches found between two value-vectors for each benchmark. The run-times obtained by executing the benchmarks by our tool (SVP) as well as by that of [95] (PE) show that the symbolic value propagation method takes less time. The crosses ($\times$) in the column corresponding to PE represents that the tool exited with false negative results in those cases since it is unable to handle code motions across loops. Other than transformations like path merging/splitting and code motions across loops, the transformations that were applied to the original behaviours to produce the corresponding optimized behaviours include associative, commutative, distributive transformations, copy and constant propagation, common subexpression elimination, arithmetic ex-

Table 3.3: Verification results based on the benchmarks presented in [90]

| Benchmarks | #BB | #Branch | #Path | #State | | Maximum | Time (ms) | |
|---|---|---|---|---|---|---|---|---|
| | | | | orig | trans | mismatch | PN | SVP |
| BARCODE-3 | 28 | 25 | 55 | 32 | 29 | 5 | 3.00E+3 | 12.2 |
| DHRC-3 | 7 | 14 | 31 | 62 | 47 | 5 | 1.62E+5 | 28.1 |
| DIFFEQ-3 | 4 | 1 | 3 | 16 | 10 | 4 | 2.69E+2 | 4.0 |
| FINDMIN8-3 | 14 | 7 | 15 | 8 | 9 | 7 | 1.00E+5 | 2.3 |
| GCD-3 | 7 | 5 | 11 | 7 | 6 | 2 | 3.25E+2 | 2.1 |
| IEEE754-3 | 40 | 28 | 59 | 55 | 42 | 6 | 3.40E+5 | 24.0 |
| LRU-3 | 22 | 19 | 39 | 33 | 25 | 4 | 3.00E+3 | 4.7 |
| MODN-3 | 7 | 4 | 9 | 6 | 5 | 4 | 5.41E+2 | 2.4 |
| PARKER-3 | 14 | 6 | 13 | 12 | 10 | 6 | 6.00E+3 | 3.1 |
| PERFECT-3 | 6 | 3 | 7 | 9 | 6 | 2 | 1.07E+2 | 1.1 |
| PRAWN-3 | 85 | 53 | 154 | 122 | 114 | 8 | 5.81E+5 | 61.5 |
| QRS-3 | 26 | 16 | 35 | 53 | 24 | 12 | 1.94E+5 | 15.7 |
| TLC-3 | 17 | 6 | 20 | 13 | 13 | 3 | 2.45E+2 | 4.0 |
| WAKA-3 | 6 | 2 | 5 | 9 | 12 | 6 | 1.00E+3 | 2.2 |

pression simplification, partial evaluation, constant (un)folding, redundant computation elimination, etc. As demonstrated in Table 3.1, the verification tool was able to establish equivalences in all the cases.

Table 3.2 gives a comparison of the execution times required by PE and SVP for the benchmarks presented in [95] which involve no code motion across loops. Moreover, the transformed FSMDs considered in [95] have been obtained by subjecting the original FSMDs to a single compiler at a time. Although the source FSMDs for all the common benchmarks in Table 3.1 and Table 3.2 are identical, the transformed FSMDs are not. Hence, to differentiate between the benchmark suites, the numerals 1 and 2 have been appended with the benchmark names. From the results it can be seen that for the benchmarks which are scheduled using a basic block (BB) based SAST [117] and the path based scheduler, SVP performs somewhat better than PE and more so for SPARK.

We further our investigation by subjecting the source codes to non-uniform code motions such as speculation, reverse speculation, safe speculation, etc. A recent work [90] addresses verification of such code motions by analyzing the data-flow of the programmes. This method resolves the decision of extending a path upon finding a mismatch for some variable, $x$ say, by checking whether the mismatched value of $x$ is used in some subsequent path or not before $x$ is defined again. For the data-flow analysis, it constructs a Kripke structure from the given behaviour, generates a CTL formula to represent the def-use relation (for $x$) and employs the model checker NuSMV [38] to find whether that formula holds in the Kripke structure or not. Depending upon the output returned by the model checker, paths are extended accordingly. However, in the presence of code motions across loop, the method of [90] fails due to the same reason as that of [95], i.e., prohibition of extension of paths beyond loops. For comparing the method of [90] with that of ours, another set of test cases have been constructed where the original behaviours have undergone both uniform and non-uniform code motions to produce the transformed behaviours. It is important to note that in none of these cases code motion across loop has been applied. The time taken by our tool and that of [90] (PN) are tabulated in Table 3.3. The method of PN takes considerable amount of more time because it spends a large proportion of its execution time interacting with NuSMV through file handling. Although a comparative analysis with the method of [115] would have been relevant, we cannot furnish it since their tool is not available to us.

## 3.8 Conclusion

In this chapter, we have presented a symbolic value propagation based equivalence checking method which not only handles control structure modifications but also code motions across loops apart from simpler uniform and non-uniform code motions without needing any supplementary information from the synthesis tools. Specifically, the contributions of the present chapter are as follows. (i) A new concept of symbolic value propagation based equivalence checking has been presented for verification of code motions across loops without compromising the capability of handling uniform and non-uniform code motions and control structure modifications achieved in earlier methods [90, 95]. (ii) The correctness of symbolic value propagation as a method of equivalence checking, and the correctness and the complexity of the equivalence checking procedure are treated formally. (iii) The computational complexity of the presented method has been analyzed and found to be no worse than that of [90]. (iv) Experimental results for several non-trivial benchmarks have been presented to empirically establish the effectiveness of the presented method. The experimental results demonstrate that our mechanism can verify equivalence between two behaviours even when one of them has been subjected to successive code transformations using the SPARK high-level synthesis tool and a path based scheduler. The method has been tested successfully on several benchmarks, including those used in [95] and [90]. The results show that the method performs comparably with both these methods in terms of verification time required and outperforms them in terms of the type of transformations handled. This method does not handle transformations such as loop unrolling, loop merging and loop shifting. Loop shifting is handled to some extent in [103, 115] at the cost of termination; however, they cannot handle control structure transformations introduced by [135] which our method can. While there are several other techniques which determine equivalence in the presence of control structure modifications, the only techniques [80, 150] which handle code motions across loops require additional information from the synthesis tools that is difficult to obtain in general. Note that while the method described in [150] captures infinite loops and computations that may fail, our method is capable of handling the former but needs some improvement to cover the latter.

# Chapter 4

# Deriving Bisimulation Relations from Path Based Equivalence Checkers

## 4.1 Introduction

For translation validation, primarily two approaches prevail namely, bisimulation based ones and path based ones; both have their own merits and demerits. As underlined in Section 1.1.2, the main drawback of bisimulation based approaches is that they require the control structures of the source and the target programs to be identical; although this limitation is alleviated to some extent in [115] using knowledge of the scheduling mechanism, more complex non-structure preserving transformations, such as those applied by [135], still remain beyond their scope. On the contrary, path based techniques [22, 90, 95] are adept in handling such non-structure preserving transformations. However, transformations such as loop shifting [44] that can be handled by bisimulation based methods of [103, 104] still elude the path based equivalence checking methods. Moreover, while the bisimulation based approaches are not guaranteed to terminate, the path based techniques are. Considering the importance of projecting equivalence of two behavioural specifications in terms of a bisimulation relation between them, in this chapter we evolve a mechanism of deriving one from the output of a path based equivalence checker.

We define the notion of simulation and bisimulation relations in the context of the FSMD model in Section 4.2. We then show in Sections 4.3 and 4.4 that a bisimulation relation can be constructed whenever two FSMDs are found to be equivalent by both a path extension based and a symbolic value propagation based equivalence checker. It is to be noted that none of the bisimulation relation based approaches has been shown to tackle code motion across loops; therefore, the present work demonstrates, for the first time, that a bisimulation relation exists even under such a context. The chapter is concluded in Section 4.5.

## 4.2 The simulation and the bisimulation relations between FSMD models

Let us consider the FSMD model $\langle Q, q_0, I, V, O, \tau : Q \times 2^S \rightarrow Q,\ h : Q \times 2^S \rightarrow U \rangle$, with the notations having their usual meanings as mentioned in Section 3.2. Let $\overline{v}$ denote a vector (an ordered tuple) of variables of $I$ and $V$ assuming values from the domain $D_{\overline{v}} = \Lambda_{v \in I \cup V} D_v$, where $D_v$ is the domain of the variable $v$ and $\Lambda$ represents the Cartesian product of sets. Let $q(\overline{v}),\ q \in Q$, denote the set of values assumed by the variables in the state $q$. Each element of $q(\overline{v})$ is called a data state at $q$ and is denoted as $\sigma_q$. Let $\Sigma = \bigcup_{q \in Q} \{\sigma_q\}$ be the set of all the data states.

An FSMD can be represented in a natural way as a digraph with the control states as the vertices and the labeled edges capturing the transition function $\tau$ and the update function $h$. Specifically, if there are two control states $q_i$ and $q_j$ such that $\tau(q_i, c(\overline{v})) = q_j$ and $h(q_i, c(\overline{v}))$ is represented as $\overline{v} \Leftarrow f(\overline{v})$, then there is an edge from $q_i$ to $q_j$ with label $c(\overline{v})/\overline{v} \Leftarrow f(\overline{v})$ in the digraph representation of the FSMD. A transition which updates a storage variable with an input is called an *input transition*. A transition which updates an output variable is called an *output transition*. A control state along with a data state together constitute a *configuration* as defined below.

**Definition 8** (Configuration). *Given an FSMD $M = \langle Q, q_0, I, V, O, \tau, h \rangle$, we define a configuration to be a pair $\langle q, \sigma \rangle$, where $q \in Q$ and $\sigma \in q(\overline{v})$.*

**Definition 9** (Execution Sequence). *For a given FSMD $M = \langle Q, q_0, I, V, O, \tau, h \rangle$, an execution sequence $\eta$ is a sequence of configurations of the form $\langle \langle q_{i_0}, \sigma_{i_0} \rangle, \langle q_{i_1}, \sigma_{i_1} \rangle, \ldots, \langle q_{i_n}, \sigma_{i_n} \rangle \rangle$, such that $\forall j, 0 \leq j \leq n, \sigma_{i_j} \in q_{i_j}(\overline{v}), \tau(q_{i_j}, c(\sigma_{i_j})) = q_{i_{j+1}}$, where $c(\sigma_{i_j})$ is true and $h(q_{i_j}, c(\sigma_{i_j})) \in U$ defines the update operation by which the data state $\sigma_{i_{j+1}}$ is obtained from $\sigma_{i_j}$.*

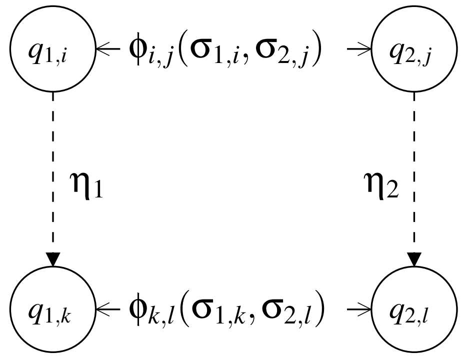

$q_{1,i}$
$\phi_{i,j}(\sigma_{1,i},\sigma_{2,j})$
$q_{2,j}$
$\eta_1$
$\eta_2$
$q_{1,k}$
$\phi_{k,l}(\sigma_{1,k},\sigma_{2,l})$
$q_{2,l}$

Figure 4.1: An example showing two execution sequences from two FSMDs.

Existence of such an execution sequence $\eta$ is denoted as $\langle q_{i_0},\sigma_{i_0}\rangle \rightsquigarrow^{\eta} \langle q_{i_n},\sigma_{i_n}\rangle$ . We denote by $\mathcal{N}_i$ the set of all execution sequences in FSMD $M_i$. From now onward we speak of two FSMDs $M_1 = \langle Q_1, q_{1,0}, I, V_1, O, \tau_1, h_1\rangle$ and $M_2 = \langle Q_2, q_{2,0}, I, V_2, O, \tau_2, h_2\rangle$ whose behaviours we seek to examine for equivalence; $V_1 \cap V_2 \neq \Phi$. Two output transitions $\langle q_{1,i}, q_{1,k}\rangle$ of $M_1$ and $\langle q_{2,j}, q_{2,l}\rangle$ of $M_2$ are said to be equivalent if they update the same output variable(s) with the same values for all execution sequences leading to them.

**Definition 10** (Equivalence of Execution Sequences)**.** *Two execution sequences $\eta_1 \in \mathcal{N}_1$ of $M_1$ and $\eta_2 \in \mathcal{N}_2$ of $M_2$ are said to be equivalent, written as $\eta_1 \equiv \eta_2$, if the two sequences contain output transitions that are pairwise equivalent.*

For the subsequent discussion, we suffix the (set of) data states in some control state of the machine $M_i, i \in \{1,2\}$, in a natural way. Similarly, we rename the variables occurring in FSMD $M_i$ by adding the suffix $i$. Thus, for machine $M_i, i \in \{1,2\}$, $\sigma_i \in \Sigma_i$ represents a data state of $M_i$ and $\sigma_{i,j} \in q_{i,j}(\overline{v_i})$ represents a data state at the control state $q_{i,j}$. A verification relation between two FSMDs $M_1$ and $M_2$ is a set of triples $\langle q_{1,i}, q_{2,j}, \phi_{i,j}\rangle$, where $q_{1,i} \in Q_1$, $q_{2,j} \in Q_2$, $\phi_{i,j} \subseteq q_{1,i}(\overline{v_1}) \times q_{2,j}(\overline{v_2})$. The notation $\phi_{i,j}(\sigma_{1,i},\sigma_{2,j}) =$ *true* indicates that $\phi_{i,j}$ is satisfied by the data states $\sigma_{1,i}$ of $M_1$ and $\sigma_{2,j}$ of $M_2$. It is important to note that $\phi_{0,0}$ is identically *true* because a computation is assumed to define always storage variables (which are later used) through some input transitions. Based on the above discussions, a simulation relation can be defined as follows (in accordance with the definition of simulation relation given in [104]).

**Definition 11** (Simulation Relation)**.** *A simulation relation $S$ for two FSMDs $M_1 = \langle Q_1, q_{1,0}, I, V_1, O, \tau_1, h_1\rangle$ and $M_2 = \langle Q_2, q_{2,0}, I, V_2, O, \tau_2, h_2\rangle$ is a verification relation which satisfies the following two clauses:*

*1.* $S(q_{1,0}, q_{2,0}, \mathit{true})$*, and*

*2.* $\forall q_{1,i}, q_{1,k} \in Q_1, \sigma_{1,i} \in q_{1,i}(\overline{v}), \sigma_{1,k} \in q_{1,k}(\overline{v}), \eta_1 \in \mathcal{N}_1, q_{2,j} \in Q_2, \sigma_{2,j} \in q_{2,j}(\overline{v})$
$\big[\ \langle q_{1,i}, \sigma_{1,i}\rangle \rightsquigarrow^{\eta_1} \langle q_{1,k}, \sigma_{1,k}\rangle \wedge \phi_{i,j}(\sigma_{1,i}, \sigma_{2,j})\ \wedge\ S(q_{1,i}, q_{2,j}, \phi_{i,j}) \Rightarrow$
$\exists q_{2,l} \in Q_2, \sigma_{2,l} \in q_{2,l}(\overline{v}), \eta_2 \in \mathcal{N}_2$
$\{\ \langle q_{2,j}, \sigma_{2,j}\rangle \rightsquigarrow^{\eta_2} \langle q_{2,l}, \sigma_{2,l}\rangle \wedge \eta_1 \equiv \eta_2\ \wedge\ \phi_{k,l}(\sigma_{1,k}, \sigma_{2,l})\ \wedge\ S(q_{1,k}, q_{2,l}, \phi_{k,l})\ \}\ \big].$

Intuitively, the two clauses mentioned above state that: (i) the reset state of FSMD $M_1$ must be related to the reset state of FSMD $M_2$ for all data states, and (ii) if two states of $M_1$ and $M_2$ with their respective data states are related and $M_1$ can proceed along an execution sequence, $\eta_1$ say, then $M_2$ must also be able to proceed along an execution sequence, $\eta_2$ say, such that $\eta_1 \equiv \eta_2$ and the end states of the two execution sequences must also be related. Figure 4.1 depicts the second clause of the definition diagrammatically. We now define a *bisimulation relation* using the definition of simulation relation.

**Definition 12** (Bisimulation Relation)**.** *A verification relation B for two FSMDs $M_1$ and $M_2$ is a bisimulation relation between them iff B is a simulation relation for $M_1, M_2$ and $B^{-1} = \{(q_{2,j}, q_{1,i}, \phi) \mid B(q_{1,i}, q_{2,j}, \phi)\}$ is a simulation relation for $M_2, M_1$.*

## 4.3 Deriving simulation relation from the output of a path extension based equivalence checker

In light of the definitions given in the previous section, let us revisit some of the earlier notions which we have come across in Section 3.2. A path $\alpha$ is characterized by an ordered pair $\langle R_\alpha, r_\alpha\rangle$, where $R_\alpha$ is the condition of execution of $\alpha$ satisfying the property that $\forall \sigma_{\alpha^s} \in \Sigma$, $R_\alpha(\sigma_{\alpha^s})$ implies that the path $\alpha$ is executed if control reaches the state $\alpha^s$; the second member $r_\alpha = \langle s_\alpha, \theta_\alpha\rangle$, where $s_\alpha$ is the functional transformation of the path $\alpha$, i.e., $\sigma_{\alpha^f} = s_\alpha(\sigma_{\alpha^s})$ and $\theta_\alpha$ represents the output list of data values of some variables produced along the path $\alpha$. The relation of corresponding states is denoted as $\delta \subseteq Q_1 \times Q_2$. Based on the above discussion, we now define a verification relation between two FSMDs.

**Definition 13** (Type-I Verification Relation)**.** *A type-I verification relation $\mathcal{V}$ for two FSMDs $M_1 = \langle Q_1, q_{1,0}, I, V_1, O, \tau_1, h_1\rangle$ and $M_2 = \langle Q_2, q_{2,0}, I, V_2, O, \tau_2, h_2\rangle$ is a verification relation which satisfies the following two clauses:*

*1.* $\mathcal{V}(q_{1,0}, q_{2,0}, true)$, *and*
*2.* $\forall q_{1,k} \in Q_1,\ q_{2,l} \in Q_2\ \big[\ \mathcal{V}(q_{1,k}, q_{2,l}, \phi_{k,l}) \Leftrightarrow$
$\exists q_{1,i} \in Q_1 (q_{1,i} \twoheadrightarrow q_{1,k} \in P_1) \wedge$
$\forall q_{1,i} \in Q_1\ \{\ q_{1,i} \twoheadrightarrow q_{1,k} = \alpha$, *say*, $\in P_1 \Rightarrow$
$\exists q_{2,j} \in Q_2\ \big(\ q_{2,j} \twoheadrightarrow q_{2,l} = \beta$, *say*, $\in P_2 \wedge \alpha \simeq \beta\ \wedge\ \mathcal{V}(q_{1,i}, q_{2,j}, \phi_{i,j})\ \big)\ \}\ \big]$.

**Theorem 5.** *The verification relation $\mathcal{V}$ for two FSMDs $M_1, M_2$ is a simulation relation for $M_1, M_2$, i.e., $\forall \langle q_{1,i}, q_{2,j}, \phi_{i,j}\rangle, \langle q_{1,i}, q_{2,j}, \phi_{i,j}\rangle \in \mathcal{V} \Rightarrow \langle q_{1,i}, q_{2,j}, \phi_{i,j}\rangle \in S$*

*Proof:* For the pair of reset states, $\langle q_{1,0}, q_{2,0}, true\rangle$ is in $\mathcal{V}$ and also in $S$ by the respective first clauses in Definition 13 and Definition 11. For $\langle q_{1,k}, q_{2,l}, \phi_{k,l}\rangle \in \mathcal{V}$, where $q_{1,k} \neq q_{1,0}$ and $q_{2,l} \neq q_{2,0}$, by application of clause 2 in Definition 13 $m$ times, $m \geq 1$, we may conclude that there exists a sequence of paths $\alpha_1, \alpha_2, \ldots, \alpha_m$, where $\alpha_h \in P_1, 1 \leq h \leq m$, and another sequence of paths $\beta_1, \beta_2, \ldots, \beta_m$, where $\beta_h \in P_2, 1 \leq h \leq m$, such that $\alpha_1^s = q_{1,0}, \alpha_m^f = q_{1,k}, \beta_1^s = q_{2,0}, \beta_m^f = q_{2,l}$ and $\alpha_h \simeq \beta_h, 1 \leq h \leq m$; also the corresponding sequence of configurations $\langle q_{1,0}, \sigma_{1,0}\rangle \xrightarrow{\alpha_1} \langle q_{1,i_1}, \sigma_{1,i_1}\rangle \xrightarrow{\alpha_2} \cdots \xrightarrow{\alpha_m} \langle q_{1,i_m} = q_{1,k}, \sigma_{1,i_m} = \sigma_{1,k}\rangle$ and $\langle q_{2,0}, \sigma_{2,0}\rangle \xrightarrow{\beta_1} \langle q_{2,j_1}, \sigma_{2,j_1}\rangle \xrightarrow{\beta_2} \cdots \xrightarrow{\beta_m} \langle q_{2,j_m} = q_{2,l}, \sigma_{2,j_m} = \sigma_{2,l}\rangle$ satisfy the property that $\langle q_{1,i_h}, q_{2,j_h}, \phi_{i_h,j_h}\rangle \in \mathcal{V}, 1 \leq h \leq m$. We shall prove that $\langle q_{1,i_h}, q_{2,j_h}, \phi_{i_h,j_h}\rangle \in S, 1 \leq h \leq m$, holds by induction on $h$.

Basis case ($h = 1$): In this case, there must exist a single path $q_{1,0} \twoheadrightarrow q_{1,k} = \alpha_1$ in $P_1$ and a path $q_{2,0} \twoheadrightarrow q_{2,l} = \beta_1$ in $P_2$ such that $\alpha_1 \simeq \beta_1$; additionally, we also have $\sigma_{1,k} = s_{\alpha_1}(\sigma_{1,0})$ and $\sigma_{2,l} = s_{\beta_1}(\sigma_{2,0})$. So, in clause 2 of Definition 11, let $q_{1,i} = q_{1,0}, \sigma_{1,i} = \sigma_{1,0}, \sigma_{1,k} = s_{\alpha_1}(\sigma_{1,0})$ so that $\sigma_{1,k} \in q_{1,k}(\bar{v}), \eta_1 = \alpha_1, q_{2,j} = q_{2,0}, \sigma_{2,j} = \sigma_{2,0}$. We notice that the antecedents of clause 2 in Definition 11 hold; specifically, the antecedent $\langle q_{1,0}, \sigma_{1,0}\rangle \rightsquigarrow^{\eta_1} \langle q_{1,k}, \sigma_{1,k}\rangle$ holds by the right hand side (rhs) of clause 2 in Definition 13; the antecedent $\phi_{0,0}(\sigma_{1,0}, \sigma_{2,0})$ holds since $\phi_{0,0}$ is identically *true*; $\langle q_{1,0}, q_{2,0}, \phi_{0,0}\rangle \in S$ by clause 1 in Definition 11. Hence, the consequents of clause 2 in Definition 11 hold with $\sigma_{2,l} = s_{\beta_1}(\sigma_{2,0}), \eta_2 = \beta_1$; so from the fourth consequent, we have $\langle q_{1,k}, q_{2,l}, \phi_{k,l}\rangle \in S$.
Induction hypothesis: Let $\langle q_{1,k}, q_{2,l}, \phi_{k,l}\rangle \in S$ hold whenever the states $q_{1,k}$ in $Q_1$ and $q_{2,l}$ in $Q_2$ are reachable from $q_{1,0}$ and $q_{2,0}$, respectively, through sequences of length $n$ of pairwise equivalent paths from $P_1$ and $P_2$.
Induction step: Let the state $q_{1,k}$ be reachable from $q_{1,0}$ through the sequence $q_{1,0} \xrightarrow{\alpha_1} q_{1,i_1} \xrightarrow{\alpha_2} \cdots \xrightarrow{\alpha_n} q_{1,i_n} \xrightarrow{\alpha_{n+1}} q_{1,k}$ in $M_1$ and the state $q_{2,l}$ be reachable from $q_{2,0}$ through the sequence $q_{2,0} \xrightarrow{\beta_1} q_{2,j_1} \xrightarrow{\beta_2} \cdots \xrightarrow{\beta_n} q_{2,j_n} \xrightarrow{\beta_{n+1}} q_{2,l}$ in $M_2$ such

**Algorithm 6** deriveSimRelVR1 (FSMD $M_1$, FSMD $M_2$, Set $\delta$)

**Inputs:** Two FSMDs $M_1$ and $M_2$, and the set $\delta$ of corresponding state pairs obtained from $M_1$ and $M_2$ by the path extension based equivalence checker.

**Outputs:** A verification relation $\mathcal{V}_1 = \{\langle q_{1,k}, q_{2,l}, \phi_{k,l}\rangle \mid \langle q_{1,k}, q_{2,l}\rangle \in \delta\}$.

```
1: Let the relation $\mathcal{V}_1$ be empty.
2: Perform live variable analyses on $M_1$ and $M_2$ and compute the set of live variables for
   each of the states that appears as a member of the state pairs in $\delta$.
3: Rename each $v_j \in V_i$ as $v_{i,j}, i \in \{1,2\}$.
4: For the pair $\langle q_{1,0}, q_{2,0}\rangle$ in $\delta$, let $\phi_{0,0}$ be $true$ and $\mathcal{V}_1 \leftarrow \mathcal{V}_1 \cup \{\langle q_{1,0}, q_{2,0}, \phi_{0,0}\rangle\}$.
5: For each of the other state pairs $\langle q_{1,i}, q_{2,j}\rangle$ in $\delta$, let $\phi_{i,j}$ be $v_{1,k_1} = v_{2,k_1} \wedge \ldots \wedge v_{1,k_n} =$
   $v_{2,k_n}$, where $v_{k_1}, \ldots, v_{k_n}$ are the common variables live at $q_{1,i}$ and $q_{2,j}$;   $\mathcal{V}_1 \leftarrow \mathcal{V}_1 \cup$
   $\{\langle q_{1,i}, q_{2,j}, \phi_{i,j}\rangle\}$.
6: **return** $\mathcal{V}_1$.
```

that $\alpha_h \simeq \beta_h, 1 \leq h \leq n+1$. Now $\langle q_{1,i_n}, q_{2,j_n}, \phi_{i_n,j_n}\rangle \in S$ by the induction hypothesis. So, in clause 2 in Definition 11, let $q_{1,i} = q_{1,i_n}, \sigma_{1,i} = \sigma_{1,i_n}, \sigma_{1,k} = s_{\alpha_{n+1}}(\sigma_{1,i_n})$ so that $\sigma_{1,k} = q_{1,k}(\overline{v}), \eta_1 = \alpha_{n+1}, q_{2,j} = q_{2,j_n}, \sigma_{2,j} = \sigma_{2,j_n}$. We notice that the antecedent $\langle q_{1,i_n}, \sigma_{1,i_n}\rangle \rightsquigarrow^{\eta_1} \langle q_{1,k}, \sigma_{1,k}\rangle$ holds by the rhs of clause 2 in Definition 13; the antecedents $\phi_{i_n,j_n}(\sigma_{1,i_n}, \sigma_{2,j_n})$ and $\langle q_{1,i_n}, q_{2,j_n}, \phi_{i_n,j_n}\rangle \in S$ hold by the induction hypothesis. Hence, the consequents of clause 2 in Definition 11 hold with $\sigma_{2,l} = s_{\beta_{n+1}}(\sigma_{2,j_n}), \eta_2 = \beta_{n+1}$; so from the fourth consequent, we have $\langle q_{1,k}, q_{2,l}, \phi_{k,l}\rangle \in S$. ■

A member of the form $\langle q_{1,i}, q_{2,j}, \phi_{i,j}\rangle$ in both type-I verification relation and simulation relation indicates that the data states at $q_{1,i}$ and $q_{2,j}$ satisfy the predicate $\phi_{i,j}$. The predicate $\phi_{i,j}$ involves the variables appearing in the two FSMDs as free variables. For path based equivalence checkers, this formula is identically *true* for the pair of reset states; for any other pair of corresponding states $\langle q_{1,i}, q_{2,j}\rangle$, it is of the form $v_{1,k_1} = v_{2,k_1} \wedge \ldots \wedge v_{1,k_n} = v_{2,k_n}$, where $v_{k_1}, \ldots, v_{k_n}$ are the common variables live at $q_{1,i}$ and $q_{2,j}$, which is precisely the criterion that must be satisfied for the two paths to be declared equivalent by a path based equivalence checker. It is also to be noted that presence of *live* uncommon variables always leads to path extension; hence, at the corresponding state pairs, there is no live uncommon variable; consequently, the uncommon variables do not appear in the $\phi_{i,j}$'s.

For establishing the fact that the path based equivalence checking method leads to a type-I verification relation which is a simulation relation between two FSMDs, we use the following notation. Let the symbol $\overline{v_c}$ represent the name vector comprising the common variables in $V_1 \cap V_2$; the vector $\overline{v_c}$ assumes values from the domain $(\Sigma_1 \cup \Sigma_2)|_{\overline{v_c}}$. Let $\overline{v_{1c}}$ and $\overline{v_{2c}}$ represent the name vectors over the common variables after renaming the components of $\overline{v_c}$ by respectively adding suffix 1 for FSMD $M_1$ and suffix 2 for FSMD $M_2$. Thus, $\overline{v_{1c}}$ ($\overline{v_{2c}}$) assumes values from the domain $\Sigma_1|_{\overline{v_c}}$ ($\Sigma_2|_{\overline{v_c}}$). We additionally use the symbol $L_{i,j}(\overline{v_i}), i \in \{1,2\}$, to denote a vector containing only those variables from $\overline{v_i}$ that are live at state $q_{i,j}$; the operation $\{\overline{e}/\overline{v}\}$ is called a substitution; the expression $\kappa\{\overline{e}/\overline{v}\}$ represents that all the occurrences of each variable $v_j \in \overline{v}$ in $\kappa$ is replaced by the corresponding expression $e_j \in \overline{e}$ simultaneously with other variables. Let $\sigma_{1,i} \in q_{1,i}(\overline{v_1})$ and $\sigma_{2,j} \in q_{2,j}(\overline{v_2})$; then $L_{1,i}(\sigma_{1,i})$ represents the values assumed by the live variables corresponding to the data state $\sigma_{1,i}$ at the control state $q_{1,i}$ in $M_1$; $L_{2,j}(\sigma_{2,j})$ is defined similarly.

**Theorem 6.** *If a path based equivalence checker declares an FSMD $M_1$ to be contained in another FSMD $M_2$, then there is a simulation relation between $M_1$ and $M_2$ corresponding to the state correspondence relation $\delta$ produced by the equivalence checker. Symbolically,*
$\forall\langle q_{1,i}, q_{2,j}\rangle \in Q_1 \times Q_2,\ \delta(q_{1,i}, q_{2,j}) \Rightarrow \exists\phi_{i,j}\ S(q_{1,i}, q_{2,j}, \phi_{i,j})$.

*Proof:* We actually prove that $\forall\langle q_{1,i}, q_{2,j}\rangle \in Q_1 \times Q_2,\ \delta(q_{1,i}, q_{2,j}) \Rightarrow \exists\phi_{i,j}\ \langle q_{1,i}, q_{2,j}, \phi_{i,j}\rangle \in \mathcal{V}$, the type-I verification relation, and then apply Theorem 5 to infer that $\langle q_{1,i}, q_{2,j}, \phi_{i,j}\rangle \in S$.
Construction of $\phi_{i,j}$'s and a verification relation $\mathcal{V}_1$: Algorithm 6 shows the steps to obtain a verification relation $\mathcal{V}_1$ from the output of a path based equivalence checker. The notation $L_{1,i}(\overline{v_{1c}}) = L_{2,j}(\overline{v_{2c}})$ implies that all the common variables that are live at $q_{1,i}$ in $M_1$ and $q_{2,j}$ in $M_2$ assume the same values at these states; symbolically, $\forall\sigma_{1,i} \in q_{1,i}(\overline{v_1}), \exists\sigma_{2,j} \in q_{2,j}(\overline{v_2}), L_{1,i}(\overline{v_{1c}})\{\sigma_{1,i}|_{\overline{v_{1c}}}/\overline{v_{1c}}\} = L_{2,j}(\overline{v_{2c}})\{\sigma_{2,j}|_{\overline{v_{2c}}}/\overline{v_{2c}}\}$. (Algorithm 6 depicts this equality more elaborately as a conjunction of equalities of the corresponding components of the vectors $L_{1,i}(\overline{v_{1c}})$ and $L_{2,j}(\overline{v_{2c}})$.) Now, we prove that the verification relation $\mathcal{V}_1$ constructed in Algorithm 6 is indeed a type-I verification relation, i.e., the relation $\mathcal{V}_1$ conforms with Definition 13.

Consider any $\langle q_{1,i}, q_{2,j}, \phi_{i,j}\rangle \in \mathcal{V}_1$. The triple $\langle q_{1,i}, q_{2,j}, \phi_{i,j}\rangle$ must have been put in $\mathcal{V}_1$ either in step 4 (case 1) or in step 5 (case 2).
For case 1, $q_{1,i} = q_{1,0}$, $q_{2,j} = q_{2,0}$ and $\phi_{i,j} = \mathit{true}$. From clause 1 of Definition 13, $\langle q_{1,0}, q_{2,0}, \mathit{true}\rangle \in \mathcal{V}$.
For case 2, from step 5 of Algorithm 6, it follows that $\langle q_{1,i}, q_{2,j}\rangle \in \delta$ and $L_{1,i}(\overline{v_{1c}}) =$

$L_{2,j}(\overline{v_{2c}})$. From Definition 6 of $\delta$, it follows that there exists a sequence of configurations of $M_1$ of the form $\langle q_{1,0}, \sigma_{1,0} \rangle \xrightarrow{\alpha_1}\!\!\!\!\rightarrow \langle q_{1,i_1}, \sigma_{1,i_1} \rangle \xrightarrow{\alpha_2}\!\!\!\!\rightarrow \langle q_{1,i_2}, \sigma_{1,i_2} \rangle \xrightarrow{\alpha_3}\!\!\!\!\rightarrow \cdots \xrightarrow{\alpha_n}\!\!\!\!\rightarrow \langle q_{1,i_n} = q_{1,i}, \sigma_{1,i_n} = \sigma_{1,i} \rangle$, where $\alpha_1, \alpha_2, \ldots, \alpha_n \in P_1$, and a sequence of configurations of $M_2$ of the form $\langle q_{2,0}, \sigma_{2,0} \rangle \xrightarrow{\beta_1}\!\!\!\!\rightarrow \langle q_{2,j_1}, \sigma_{2,j_1} \rangle \xrightarrow{\beta_2}\!\!\!\!\rightarrow \langle q_{2,j_2}, \sigma_{2,j_2} \rangle \xrightarrow{\beta_3}\!\!\!\!\rightarrow \cdots \xrightarrow{\beta_n}\!\!\!\!\rightarrow \langle q_{2,j_n} = q_{2,j}, \sigma_{2,j_n} = \sigma_{2,j} \rangle$, where $\beta_1, \beta_2, \ldots, \beta_n \in P_2$, such that $\alpha_h \simeq \beta_h$ and $\langle q_{1,i_h}, q_{2,j_h} \rangle \in \delta, 1 \le h \le n$. Hence, from step 5 of Algorithm 6, $\langle q_{1,i_h}, q_{2,j_h}, \phi_{i_h,j_h} \rangle \in \mathcal{V}_1, 1 \le h \le n$. We show that $\langle q_{1,i_h}, q_{2,j_h}, \phi_{i_h,j_h} \rangle \in \mathcal{V}, 1 \le h \le n$, by induction on $h$.

*Basis ($h = 1$):* Let us consider the rhs of the biconditional ($\Leftrightarrow$) in clause 2 of Definition 13 with $q_{1,k} = q_{1,i_1}$ and $q_{2,l} = q_{2,j_1}$. Let $q_{1,i} = q_{1,0}$; the first conjunct $q_{1,i} \twoheadrightarrow q_{1,i_1} \in P_1$ holds because $q_{1,0} \twoheadrightarrow q_{1,k} = \alpha_1 \in P_1$. The second conjunct holds with $q_{1,i} = q_{1,0}, q_{2,j} = q_{2,0}$, $q_{2,0} \twoheadrightarrow (q_{2,k} = q_{2,j_1}) = \beta_1 \in P_2$ and $\alpha_1 \simeq \beta_1$ and $\langle q_{1,0}, q_{2,0}, \phi_{0,0} \rangle \in \mathcal{V}$ from clause 1 of Definition 13. Hence, the lhs of the biconditional in clause 2 of Definition 13 holds yielding $\langle q_{1,i_1}, q_{2,j_1}, \phi_{i_1,j_1} \rangle \in \mathcal{V}$.

*Induction hypothesis:* Let $\langle q_{1,i_h}, q_{2,j_h}, \phi_{i_h,j_h} \rangle \in \mathcal{V}, 1 \le h \le m < n$.

*Induction step ($h = m+1$):* Let $q_{1,k} = q_{1,i_{m+1}}, q_{2,l} = q_{2,j_{m+1}}$ in clause 2 of Definition 13. From the rhs of the biconditional in clause 2, let $q_{1,i} = q_{1,i_m}$; the first conjunct in the rhs namely, $q_{1,i} \twoheadrightarrow q_{1,i_{m+1}} = \alpha_{m+1} \in P_1$, holds with $q_{1,i} = q_{1,i_m}$. In the second conjunct of the rhs, with $q_{1,i} = q_{1,i_m}$, the antecedent $q_{1,i} \twoheadrightarrow q_{1,k} = q_{1,i_m} \twoheadrightarrow q_{1,i_{m+1}} = \alpha_{m+1} \in P_1$ holds. In the consequent of this conjunct, let $q_{2,j} = q_{2,j_m}$; we find $q_{2,j} \twoheadrightarrow q_{2,l} = q_{2,j_m} \twoheadrightarrow q_{2,j_{m+1}} = \beta_{m+1} \in P_2$ holds; $\alpha_{m+1} \simeq \beta_{m+1}$ holds and $\langle q_{1,i}, q_{2,j}, \phi_{i,j} \rangle = \langle q_{1,i_m}, q_{2,j_m}, \phi_{i_m,j_m} \rangle \in \mathcal{V}$ by induction hypothesis. So, the rhs of the biconditional holds. Hence, the lhs holds yielding $\langle q_{1,i_{m+1}}, q_{2,j_{m+1}}, \phi_{i_{m+1},j_{m+1}} \rangle \in \mathcal{V}$. ■

Note that the path extension based equivalence checker ensures $M_1 \sqsubseteq M_2$ first and then $M_2 \sqsubseteq M_1$. In the second step, it does not have to change the set $\delta$ of corresponding state pairs constructed during the first step. Hence, the simulation relation $\mathcal{V}_1$ obtained from Algorithm 6 will be a bisimulation relation too by Definition 12.

It is important to note that as per Definition 11 of simulation relation, there is no restriction on the end states of execution sequences. Algorithm 6, however, produces a bisimulation relation comprising triples for only the corresponding states. This does not impair generality because if one is interested in enhancing the bisimulation relation by incorporating triples for some control states other than corresponding states, one may easily do so by applying Dijkstra's weakest precondition computation starting from the corresponding states appearing immediately next to one's states of choice in the control flow graph of FSMD. An analogous situation arises in Kundu et al.'s method described in [103, 104], where the bisimulation relation produced comprises triples only for the *pairs of interest* which are basically pairs of control states, one from the specification and the other from the implementation (similar to our corresponding states).

## 4.4 Deriving simulation relation from the output of a symbolic value propagation based equivalence checker

The basic method of symbolic value propagation [22] consists in identifying the mismatches in the (symbolic) values of the live variables at the end of two paths taken from two different FSMDs; if mismatches in the values of some live variables are detected, then the variable values (stored as a vector) are propagated through all the subsequent path segments. Repeated propagation of values is carried out until an equivalent path or a final path segment ending in the reset state is reached. In the latter case, any prevailing discrepancy in values indicates that the original and the transformed behaviours are not equivalent; otherwise they are. Note that the conditionally corresponding (C-corresponding) states are captured by the relation $\delta_c$ (in contrast to the (unconditional) state correspondence relation $\delta$).

It has already been mentioned in the previous chapter that paths cannot be extended beyond a loop by definition and therefore pure path based approaches fail in the face of code motions across loops; the primary motivation behind developing symbolic value propagation based technique was to overcome this limitation. So, let us revisit the example shown in Figure 3.3 on page 41 which exhibits a case of code motion across loop. In this example, the verification relation tuple obtained for the state pair $\langle q_{1,1}, q_{2,1} \rangle$ is $\langle q_{1,1}, q_{2,1}, \phi_{1,1} \rangle$, where $\phi_{1,1}$ is $\exists t1'_1, t2'_1\ \exists t1'_2, t2'_2\ [x_1 = x_2 \wedge i_1 = i_2 \wedge N_1 = N_2 \wedge t1_1 = t1_2 \wedge t2_1 = t2_2 \wedge y_1 = t1'_1 - t2'_1 \wedge h_2 = t1'_2 + t2'_2]$ where the quantified variables $t1'_1, t2'_1$ represent the symbolic values of the variables $t1_1, t2_1$ at $q_{1,0}$ and $t1'_2, t2'_2$ represent the symbolic values of the variables $t1_2, t2_2$ at $q_{2,0}$ as captured by the propagated vectors for this pair of states. It is important to note that the loop $q_{1,1} \twoheadrightarrow q_{1,1}$ is executed as many times in $M_1$ as the loop $q_{2,1} \twoheadrightarrow q_{2,1}$ in $M_2$ and the above mentioned relation is maintained between the data states at $\langle q_{1,1}, q_{2,1} \rangle$ across all these executions (zero or more times). To ensure that the two loops in the two FSMDs are executed equal number of times, the conditions of execution of the loops must be identical; thus, obviously, no mismatched variable appears in the condition of execution of either of the loops.

The above discussion exhibits that in the presence of code motions across loops, establishing a verification relation to be a simulation relation would necessitate moving through several path segments with some of them arising from a loop. Hence, we resort to the level of *walks* which are sequences of states and intermediary edges having possible repetitions of states (which mark the entry/exit of a loop) as defined below.

**Definition 14** (Walk). *For a set $P$ of paths of an FSMD $M$, a walk between two states $q_i$ and $q_k$ of $M$, represented as $q_i \rightsquigarrow_P q_k$ ($= \xi$, say), is a finite sequence of* distinct paths *of $P$ of the form $\langle \alpha_1 : q_i \twoheadrightarrow q_{m_1}, \alpha_2 : q_{m_1} \twoheadrightarrow q_{m_2}, \ldots, \alpha_{l+1} : q_{m_l} \twoheadrightarrow q_k \rangle$; the condition of execution of a walk $\xi$ is given by $R_\xi = R_{\alpha_1}(\overline{v}) \wedge R_{\alpha_2}(s_{\alpha_1}(\overline{v})) \wedge R_{\alpha_3}(s_{\alpha_2}(s_{\alpha_1}(\overline{v}))) \wedge \ldots \wedge R_{\alpha_{l+1}}(s_{\alpha_l}(s_{\alpha_{l-1}} \ldots (s_{\alpha_1}(\overline{v}))\ldots))$; the functional transformation $s_\xi$ is given by $s_{\alpha_{l+1}}(s_{\alpha_l} \ldots (s_{\alpha_1}(\overline{v}))\ldots)$ and the output list $\theta_\xi$ is given by the concatenation of the following output lists of the individual constituent paths $\theta_{\alpha_1}(\overline{v})$, $\theta_{\alpha_2}(s_{\alpha_1}(\overline{v}))$, $\theta_{\alpha_3}(s_{\alpha_2}(s_{\alpha_1}(\overline{v})))$, $\ldots$, $\theta_{\alpha_{l+1}}(s_{\alpha_l}(s_{\alpha_{l-1}} \ldots (s_{\alpha_1}(\overline{v}))\ldots))$; the latter two path characterizations together comprise $r_\xi$, i.e., $r_\xi = \langle s_\xi, \theta_\xi \rangle$.*

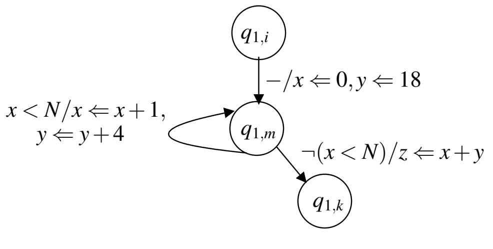
**Definition 15** (Equivalence of Walks). *Two walks $\xi$ and $\zeta$ are said to be computationally equivalent, denoted as $\xi \simeq \zeta$, iff $R_\xi \equiv R_\zeta$ and $r_\xi = r_\zeta$.*

Next we re-define the notion of corresponding states based on equivalent walks rather than paths as given in the earlier Definition 6.

**Definition 16** (Corresponding States). *Let $M_1 = \langle Q_1, q_{1,0}, I, V_1, O, \tau_1, h_1 \rangle$ and $M_2 = \langle Q_2, q_{2,0}, I, V_2, O, \tau_2, h_2 \rangle$ be two FSMDs having identical input and output sets, $I$ and $O$, respectively.*

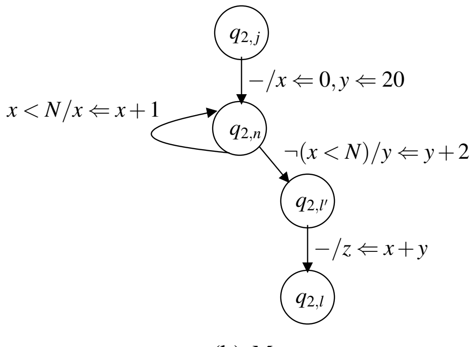
1) *The respective reset states $q_{1,0}$ and $q_{2,0}$ are corresponding states and a non-reset state does not have correspondence with a reset state.*

2) *If $q_{1,i} \in Q_1$ and $q_{2,j} \in Q_2$ are corresponding states and there exist $q_{1,k} \in Q_1$ and $q_{2,l} \in Q_2$ such that, for some walk $\xi$ from $q_{1,i}$ to $q_{1,k}$ in $M_1$, there exists a walk*



$q_{1,i}$
$-/x \Leftarrow 0, y \Leftarrow 18$
$x < N/x \Leftarrow x+1,$
$y \Leftarrow y+4$
$q_{1,m}$
$\neg(x < N)/z \Leftarrow x+y$
$q_{1,k}$

(a) $M_1$



(b) $M_2$

Figure 4.2: An example of non-equivalent programs.

*$\zeta$ from $q_{2,j}$ to $q_{2,l}$ in $M_2$ such that $\xi \simeq \zeta$, then $q_{1,k}$ and $q_{2,l}$ are corresponding states (note that $q_{1,k}$ and $q_{2,l}$ must both be either reset states or non-reset states).*

In addition to the set of corresponding states $\delta$, we also maintain the set of *corresponding loop states* to store pairs of the entry/exit states of those loops which are reached with loop invariant propagated vectors, such as $\langle q_{1,1}, q_{2,1} \rangle$ in Figure 3.3; let the set of such corresponding loop states be denoted as $\hat{\delta}$.

Let us now briefly revisit the path extension based equivalence checking method as mentioned in Section 4.3. In this method, we had a path cover $P$ of an FSMD $M$ such that every computation of $M$ could be looked upon as a concatenation of paths (possibly, with repetitions) from the path cover $P$. Note that in the present scenario, we cannot define a similar notion of *walk cover* since a walk cover may have to accommodate infinite number of walks to capture arbitrary number of executions of loops. The walks that are considered by our symbolic value propagation based equivalence checker does not allow for repetition of paths (as given in Definition 14); if a walk contains any subsequence (of paths) with identical start and end states, then that subsequence represents a loop. Let $\xi$ be a walk in FSMD $M$ of the form $\xi_p \xi_l \xi_s$ with a prefix sequence $\xi_p$ ("p" for prefix), a sequence $\xi_l$ ("l" for loop) with identical start and end state and a suffix sequence $\xi_s$ ("s" for suffix); we call a walk such as $\xi$ as a *single loop walk (SLW)*. Walks containing SLWs where the loop segments are entered with mismatches in variables can have equivalent walks in the other FSMD provided the mismatches do not change over the loop iterations as illustrated below.

In Figure 4.2(a), let the path $q_{1,i} \twoheadrightarrow q_{1,m}$ be $\xi_p$, the loop $q_{1,m} \twoheadrightarrow q_{1,m}$ be $\xi_l$ and the path $q_{1,m} \twoheadrightarrow q_{1,k}$ be $\xi_s$. Similarly, in Figure 4.2(b), let the path $q_{2,j} \twoheadrightarrow q_{2,n}$ be $\zeta_p$, the loop $q_{2,n} \twoheadrightarrow q_{2,n}$ be $\zeta_l$ and the path $q_{2,n} \twoheadrightarrow q_{2,l}$ be $\zeta_s$. It is important to note that the SLWs $\xi_p\xi_l\xi_s$ and $\zeta_p\zeta_l\zeta_s$ are equivalent having identical condition of execution $R_\xi \equiv R_\zeta \equiv (0 < N) \wedge \neg(1 < N)$ and identical data transformations $s_\xi = s_\zeta = \{x \Leftarrow 1, y \Leftarrow 22, z \Leftarrow 23\}$ for $N = 1$. However, for any integer value of $N$ other than 1, the computations of the two programs will be different. For example, for $N = 5$, the source program will assign the value 43 to the variable $z$, whereas, the transformed program will assign the value 27 to $z$.

Note that the symbolic value propagation based equivalence checker identifies two such SLWs, $\xi$ of $M_1$ and $\zeta$ of $M_2$, as equivalent (or more specifically, starting with the corresponding states $\langle \xi^s, \zeta^s \rangle$, identifies $\langle \xi^f, \zeta^f \rangle$ as corresponding states) if there are propagation vectors $\gamma_1$ and $\gamma_2$, say, at $\xi_p^f$ and $\zeta_p^f$ respectively depicting mismatches over $\xi_p$ and $\zeta_p$ which satisfy the following three conditions:
(i) $\gamma_1$ and $\gamma_2$ disappear (i.e., all mismatches are compensated for) over the segments $\xi_p^f(= \xi_s^s)$ to $\xi_s^f$ and $\zeta_p^f(= \zeta_s^s)$ to $\zeta_s^f$,
(ii) $\gamma_1$ and $\gamma_2$ remain invariant over the loops $\xi_l$ and $\zeta_l$, respectively, and
(iii) the conditions of execution $\xi_l$ and $\zeta_l$ are identical (involving no variable which has mismatch as per $\gamma_1$ and $\gamma_2$).

Conditions (i) and (ii) indicate that code motions may have taken place from $\xi_p$ to $\xi_s$ across the loop $\xi_l$, or from $\zeta_p$ to $\zeta_s$ across the loop $\zeta_l$ or both. Condition (iii) implies that the loops $\xi_l$ and $\zeta_l$ are executed identical number of times. Only under these conditions, $\xi \simeq \zeta \Rightarrow \xi_p(\xi_l)^m\xi_s \simeq \zeta_p(\zeta_l)^m\zeta_s$, $\forall m \geq 0$, where $m$ represents the number of times the loops $\xi_l$ and $\zeta_l$ are iterated. Stated in words, the loop invariance of the propagated vectors at the entry/exit states of the loop preserves equivalence of SLWs under pumping of the loops equal number of times. The following lemma supports the observation.

**Lemma 1** (Pumping Lemma for SLWs). *Let $\xi = \xi_p\xi_l\xi_s$ be an SLW of $M_1$ and $\zeta = \zeta_p\zeta_l\zeta_s$ be an SLW of $M_2$ such that $\langle \xi^s, \zeta^s \rangle \in \delta$; let $\gamma_1$ and $\gamma_2$ be the propagation vectors over $\xi_p$ and $\zeta_p$, respectively. If $\gamma_1$ and $\gamma_2$ are loop invariant over $\xi_l$ and $\zeta_l$, respectively, then $\xi \simeq \zeta \Rightarrow \xi_p(\xi_l)^m\xi_s \simeq \zeta_p(\zeta_l)^m\zeta_s$, $\forall m \geq 0$.*

*Proof:* The constraint $\langle \xi^s, \zeta^s \rangle \in \delta$ implies that all the (common) live variables must have identical values at $\xi^s$ in $M_1$ and $\zeta^s$ in $M_2$. Let $v$ be a variable whose definitions mismatch at $\xi_p^f(= \xi_l^s)$ and $\zeta_p^f(= \zeta_l^s)$. Since the propagated vectors $\gamma_1$ and $\gamma_2$ are loop invariant, the variables which have the mismatches at $\xi_l^s$ and $\zeta_l^s$ as well as those which appear in their symbolic expression values are not assigned any new values and the other variables are transformed identically. In short, therefore, $\mathsf{s}_{\xi_l} = \mathsf{s}_{\zeta_l}$. Also, the conditions of execution of the loops, $R_{\xi_l}$ and $R_{\zeta_l}$, are equivalent and do not involve any variables which have mismatches at $\xi_l^s$ and $\zeta_l^s$. Under this scenario, the loops $\xi_l$ and $\zeta_l$ will iterate identical number of times, $\mathsf{s}_{\xi_l}^m(\mathsf{s}_{\xi_p}(\overline{v_1}))$ and $\mathsf{s}_{\zeta_l}^m(\mathsf{s}_{\zeta_p}(\overline{v_2}))$ will have the same mismatch and $R_{\xi_l}(\mathsf{s}_{\xi_l}^m(\mathsf{s}_{\xi_p}(\overline{v_1}))) \equiv R_{\zeta_l}(\mathsf{s}_{\zeta_l}^m(\mathsf{s}_{\zeta_p}(\overline{v_2}))), \forall m \geq 0$. It further implies that mismatched variables, such as $v$, must not have been output in $\xi_l$ and $\zeta_l$, and also not used in determining the value of any other variable. Finally, the antecedent $\xi \simeq \zeta$ implies that the mismatch in definitions must have disappeared at $\langle \xi^f, \zeta^f \rangle$; hence, $R_{\xi_p}(\overline{v_1}) \wedge R_{\xi_l}(\mathsf{s}_{\xi_p}(\overline{v_1})) \wedge R_{\xi_s}(\mathsf{s}_{\xi_l}(\mathsf{s}_{\xi_p}(\overline{v_1}))) \equiv R_{\zeta_p}(\overline{v_2}) \wedge R_{\zeta_l}(\mathsf{s}_{\zeta_p}(\overline{v_2})) \wedge R_{\zeta_s}(\mathsf{s}_{\zeta_l}(\mathsf{s}_{\zeta_p}(\overline{v_2})))$ and $\mathsf{s}_{\xi_s}(\mathsf{s}_{\xi_l}(\mathsf{s}_{\xi_p}(\overline{v_1}))) = \mathsf{s}_{\zeta_s}(\mathsf{s}_{\zeta_l}(\mathsf{s}_{\zeta_p}(\overline{v_2})))$. Therefore, from $R_{\xi_l} \equiv R_{\zeta_l}$ and $\mathsf{s}_{\xi_l} = \mathsf{s}_{\zeta_l}$, it follows that $R_{\xi_p}(\overline{v_1}) \wedge R_{\xi_l}(\mathsf{s}_{\xi_p}(\overline{v_1})) \wedge R_{\xi_l}(\mathsf{s}_{\xi_l}(\mathsf{s}_{\xi_p}(\overline{v_1}))) \wedge R_{\xi_s}(\mathsf{s}_{\xi_l}(\mathsf{s}_{\xi_l}(\mathsf{s}_{\xi_p}(\overline{v_1})))) \equiv R_{\zeta_p}(\overline{v_2}) \wedge R_{\zeta_l}(\mathsf{s}_{\zeta_p}(\overline{v_2})) \wedge R_{\zeta_l}(\mathsf{s}_{\zeta_l}(\mathsf{s}_{\zeta_p}(\overline{v_2}))) \wedge R_{\zeta_s}(\mathsf{s}_{\zeta_l}(\mathsf{s}_{\zeta_l}(\mathsf{s}_{\zeta_p}(\overline{v_2}))))$ and $\mathsf{s}_{\xi_s}(\mathsf{s}_{\xi_l}(\mathsf{s}_{\xi_l}(\mathsf{s}_{\xi_p}(\overline{v_1})))) = \mathsf{s}_{\zeta_s}(\mathsf{s}_{\zeta_l}(\mathsf{s}_{\zeta_l}(\mathsf{s}_{\zeta_p}(\overline{v_2}))))$; repetition of the argument $m-1$ times gives $R_{\xi_p}(\overline{v_1}) \wedge R_{\xi_l}(\mathsf{s}_{\xi_l}^{m-1}( \mathsf{s}_{\xi_p}(\overline{v_1}))) \wedge R_{\xi_s}(\mathsf{s}_{\xi_l}^m(\mathsf{s}_{\xi_p}(\overline{v_1}))) \equiv R_{\zeta_p}(\overline{v_2}) \wedge R_{\zeta_l}(\mathsf{s}_{\zeta_l}^{m-1}(\mathsf{s}_{\zeta_p}(\overline{v_2}))) \wedge R_{\zeta_s}(\mathsf{s}_{\zeta_l}^m(\mathsf{s}_{\zeta_p}(\overline{v_2})))$ and $\mathsf{s}_{\xi_s}(\mathsf{s}_{\xi_l}^m(\mathsf{s}_{\xi_p}(\overline{v_1}))) = \mathsf{s}_{\zeta_s}(\mathsf{s}_{\zeta_l}^m(\mathsf{s}_{\zeta_p}(\overline{v_2})))$; thus, $\xi_p.(\xi_l)^m.\xi_s \simeq \zeta_p.(\zeta_l)^m.\zeta_s, m \geq 1$. Again from the invariance of $\gamma_1$ over the loop $\xi_l$ and that of $\gamma_2$ over the loop $\zeta_l$ and the antecedent $\xi \simeq \zeta$, it follows that $R_{\xi_p}(\overline{v_1}) \wedge R_{\xi_s}(\mathsf{s}_{\xi_p}(\overline{v_1})) \equiv R_{\zeta_p}(\overline{v_2}) \wedge R_{\zeta_s}(\mathsf{s}_{\zeta_p}(\overline{v_2}))$ and $\mathsf{s}_{\xi_s}(\mathsf{s}_{\xi_p}(\overline{v_1})) = \mathsf{s}_{\zeta_s}(\mathsf{s}_{\zeta_p}(\overline{v_2}))$. Thus, $\xi \simeq \zeta \Rightarrow \xi_p(\xi_l)^m\xi_s \simeq \zeta_p(\zeta_l)^m\zeta_s$, $\forall m \geq 0$. ■

Henceforth, the term SLW is used to mean an SLW that satisfies the property of loop invariance, if not explicitly stated otherwise. With this concept of SLWs, a walk cover can now be defined as follows.

**Definition 17** (Walk Cover of an FSMD). *A finite set of walks $W = \{\xi_1, \xi_2, \ldots, \xi_n\}$ (for a path cover $P$) is said to be a walk cover of an FSMD $M$ if any computation of $M$ can be represented as a sequence of the form $\rho_1\rho_2\ldots\rho_k$ which satisfies the following conditions:*

*(i)* $\forall i, 1 \leq i \leq k$,

*(a)* $\rho_i \in W$, *or*

*(b)* $\exists \xi \in W$ *of the form* $\xi_p\xi_l\xi_s$ *such that* $\rho_i = \xi_p$ *or* $\rho_i = \xi_l$ *or* $\rho_i = \xi_s$ *and if* $\rho_i$ *occurs in the computation consecutively more than once, then* $\rho_i = \xi_l$,

*(ii)* $\rho_1^s$ *and* $\rho_k^f$ *are the reset state of* $M$.

Now we have the following theorem which validates our symbolic value propagation based equivalence checking method around walks as given in Definition 14.

**Theorem 7.** *An FSMD $M_1$ is contained in another FSMD $M_2$ ($M_1 \sqsubseteq M_2$), if there exists a finite walk cover $W_1 = \{\xi_1, \xi_2, \ldots, \xi_{l_1}\}$ of $M_1$ for which there exists a set of walks $W_2 = \{\zeta_1, \zeta_2, \ldots, \zeta_{l_2}\}$ of $M_2$ such that for any corresponding state pair $\langle q_{1,i}, q_{2,j} \rangle$, for any walk $\xi_m \in W_1$ emanating from $q_{1,i}$, there exists a walk $\zeta_n \in W_2$ emanating from $q_{2,j}$ such that $\xi_m \simeq \zeta_n$.*

*Proof:* From Definition 2, $M_1 \sqsubseteq M_2$, if for any computation $\mu_1$ of $M_1$ on some inputs, there exists a computation $\mu_2$ of $M_2$ on the same inputs such that $\mu_1 \simeq \mu_2$. Since $W_1$ is a walk cover of $M_1$ (as per Definition 17), the computation $\mu_1$ of $M_1$ can be represented in terms of walks from $W_1$ starting from the reset state $q_{1,0}$ and ending again at this reset state of $M_1$. Consider a computation $\mu_1$ of the form $[\xi_{k_1}\xi_{k_2}\cdots\xi_{k_t}]$ where $\xi_{k_i} \in W_1, \forall i, 1 \leq i \leq t$. We have to show that a computation $\mu_2$ exists in $M_2$ such that $\mu_1 \simeq \mu_2$.

In general, $\mu_1$ may contain multiple iterations of various loops in $M_1$. Note that cut-point introduction rules ensure that loops have identical entry and exit state. We may have the following two cases:
*Case 1:* The entry state of a loop $l_1$, say, in $M_1$ has correspondence $\delta$ with the entry state of some loop $l_2$ in $M_2$. Under this case, $l_1$ and $l_2$ are designated as walks in the two FSMDs as per Definition 16.
*Case 2:* The entry state of a loop in $M_1$ has no correspondence $\delta$ with the entry state of some loop in $M_2$; this happens when there is mismatch of values of some variables that resulted either prior to or within the corresponding loop segments in the two FSMDs. If the mismatch originates in the loop segment then the hypothesis of the theorem does not hold. So we only consider the scenarios where the mismatch originates prior to entry to the corresponding loop segments. The existence of walk cover $W_1$ ensures that the mismatch originated in some segment leading to the loop remains invariant over the loop and disappears over the segment following the loop. In other words, $W_1$ contains a walk of the form $\xi_p\xi_l\xi_s$.

Before we can mechanically obtain a computation $\mu_2$ of $M_2$, such that $\mu_1 \simeq \mu_2$, it is to be noted that $\mu_1$ may contain multiple iterations (or zero iteration) of a loop $\xi_l$ over which propagated vectors, if any, have remained invariant; subsequences of the

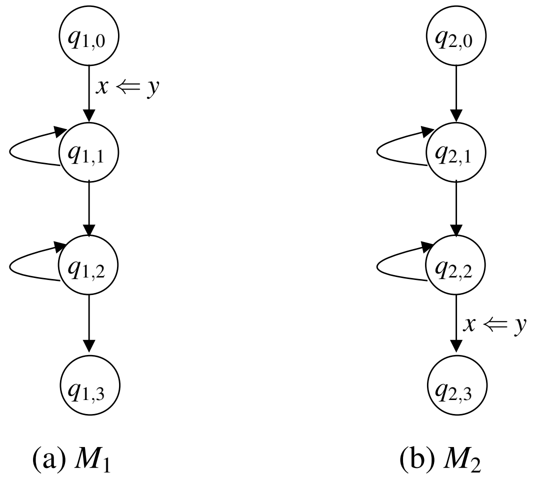


Figure 4.3: An example of code motion across multiple loops.

form $\xi_p(\xi_l)^d\xi_s, d \geq 0$, are replaced by the SLW $\xi_p\xi_l\xi_s$. After all such substitutions have been carried out, let the modified computation $\mu_1'$ be $[\xi_{i_1}\xi_{i_2}\ldots\xi_{i_{t'}}]$.

The reset states $q_{1,0}$ of $M_1$ and $q_{2,0}$ of $M_2$ must be corresponding states by clause 1 of Definition 16. Therefore, it follows from the hypothesis that a walk $\zeta_{j_1}$ exists in $W_2$ such that $\xi_{i_1} \simeq \zeta_{j_1}$; if $\xi_{i_1}$ is an SLW, then so shall be $\zeta_{j_1}$; otherwise, both will be simple walks (without loops). Thus, the states $\xi_{i_1}^f$ and $\zeta_{j_1}^f$ must again be corresponding states by clause 2 in Definition 16. By repetitive applications of the above argument, it follows that there exists a concatenated sequence of paths $\zeta_{j_1}\ldots\zeta_{j_t}$ such that $\xi_{i_k} \simeq \zeta_{j_k}, 1 \leq k \leq t$. What remains to be proved for $[\zeta_{j_1}\zeta_{j_2}\ldots\zeta_{j_t}]$ to be a computation of $M_2$ is that $\zeta_{j_t}^f = q_{2,0}$. Let $\zeta_{j_t}^f \neq q_{2,0}$; now $\langle\xi_{i_t}^f, \zeta_{j_t}^f\rangle$, i.e., $\langle q_{1,0}, \zeta_{j_t}^f\rangle$ must be a corresponding state pair. However, by Definition 16, a non-reset state cannot have correspondence with a reset state. Consequently, $\zeta_{j_t}^f$ must be $q_{2,0}$ and thus $[\zeta_{j_1}\zeta_{j_2}\ldots\zeta_{j_t}]$ is a computation, $\mu_2'$ say, and $\mu_1' \simeq \mu_2'$. In order to obtain the intended computation $\mu_2$ from $\mu_2'$, we introduce in $\mu_2'$ as many iterations of the loop $\xi_l$ as there are for the equivalent loop $\zeta_l$ in $\mu_1$, i.e., we essentially perform a substitution on $\mu_2'$ which is just the reverse of the substitution that was applied on $\mu_1$ to obtain $\mu_1'$. It follows from Lemma 1, that the newly obtained $\mu_2$ is equivalent to $\mu_1$. ■

It is crucial to note that our symbolic value propagation based equivalence checker can establish equivalence even when some code motion has taken place across multiple loops as shown in Figure 4.3 (considering $x$ has not been output or used and $y$ has not been updated in the paths $q_{1,1} \twoheadrightarrow q_{1,1}$, $q_{1,1} \twoheadrightarrow q_{1,2}$ and $q_{1,2} \twoheadrightarrow q_{1,2}$). Thus, although the formalism discussed so far deals with single loop walks, it can be easily extended to accommodate multiple loop walks; for this, the definition of walk cover (Definition 17), its corresponding Lemma 1, Theorem 7 and their proofs can be suitably extended. Note that the definition of walk given in Definition 14 requires no modification to incorporate the notion of multiple loop walks.

Next, to establish that the symbolic value propagation based equivalence checker yields a simulation relation, we define, in the following, another verification relation accommodating the notion of walks (as given in Definition 14).

**Definition 18** (Type-II Verification Relation). *A type-II verification relation $\mathcal{V}'$ for two FSMDs $M_1 = \langle Q_1, q_{1,0}, I, V_1, O, \tau_1, h_1\rangle$ and $M_2 = \langle Q_2, q_{2,0}, I, V_2, O, \tau_2, h_2\rangle$ is a verification relation which satisfies the following two clauses:*
*1. $\mathcal{V}'(q_{1,0}, q_{2,0}, true)$, and*
*2. $\forall q_{1,k} \in Q_1,\ q_{2,l} \in Q_2\ \big[\ \mathcal{V}'(q_{1,k}, q_{2,l}, \phi_{k,l}) \Leftrightarrow$*
*$\exists q_{1,i} \in Q_1 (q_{1,i} \rightsquigarrow_{P_1} q_{1,k}) \wedge$*
*$\forall q_{1,i} \in Q_1\ \big\{\ q_{1,i} \rightsquigarrow_{P_1} q_{1,k} = \xi$, say, $\Rightarrow$*
*$\exists q_{2,j} \in Q_2\ \big(\ q_{2,j} \rightsquigarrow_{P_2} q_{2,l} = \zeta$, say, $\wedge\, \xi \simeq \zeta\ \wedge\ \mathcal{V}'(q_{1,i}, q_{2,j}, \phi_{i,j})\ \big)\ \big\}\ \big]$.*

Note that if we allow all walks to be only single paths from the path cover, then Definition 18 boils down to Definition 13, i.e., the latter is a specific case of the more general Definition 18.

**Theorem 8.** *The verification relation $\mathcal{V}'$ for two FSMDs $M_1, M_2$ is a simulation relation for $M_1, M_2$, i.e., $\forall \langle q_{1,i}, q_{2,j}, \phi_{i,j}\rangle, \langle q_{1,i}, q_{2,j}, \phi_{i,j}\rangle \in \mathcal{V}' \Rightarrow \langle q_{1,i}, q_{2,j}, \phi_{i,j}\rangle \in S$.*

*Proof:* For the pair of reset states, $\langle q_{1,0}, q_{2,0}, true\rangle$ is in $\mathcal{V}'$ and also in $S$ by the respective first clauses in Definition 18 and Definition 11. For $\langle q_{1,k}, q_{2,l}, \phi_{k,l}\rangle \in \mathcal{V}'$, where $q_{1,k} \neq q_{1,0}$ and $q_{2,l} \neq q_{2,0}$, by application of clause 2 in Definition 18 in the forward direction (i.e., along $\Rightarrow$) $m$ times, $m \geq 1$, we may conclude that there exists a sequence of walks $\xi_1, \xi_2, \ldots, \xi_m$, where the paths constituting $\xi_h, 1 \leq h \leq m$, all belong to $P_1$ and another sequence of walks $\zeta_1, \zeta_2, \ldots, \zeta_m$, where the paths constituting $\zeta_h, 1 \leq h \leq m$, all belong to $P_2$ such that $\xi_1^s = q_{1,0}, \xi_m^f = q_{1,k}, \zeta_1^s = q_{2,0}, \zeta_m^f = q_{2,l}$ and $\xi_h \simeq \zeta_h, 1 \leq h \leq m$; also the corresponding sequence of configurations $\langle q_{1,0}, \sigma_{1,0}\rangle \rightsquigarrow^{\xi_1} \langle q_{1,i_1}, \sigma_{1,i_1}\rangle \rightsquigarrow^{\xi_2} \cdots \rightsquigarrow^{\xi_m} \langle q_{1,i_m} = q_{1,k}, \sigma_{1,i_m} = \sigma_{1,k}\rangle$ and $\langle q_{2,0}, \sigma_{2,0}\rangle \rightsquigarrow^{\zeta_1} \langle q_{2,j_1}, \sigma_{2,j_1}\rangle \rightsquigarrow^{\zeta_2} \cdots \rightsquigarrow^{\zeta_m} \langle q_{2,j_m} = q_{2,l}, \sigma_{2,j_m} = \sigma_{2,l}\rangle$ satisfy the property that $\langle q_{1,i_h}, q_{2,j_h}, \phi_{i_h,j_h}\rangle \in \mathcal{V}', 1 \leq h \leq m$. We shall prove that $\langle q_{1,i_h}, q_{2,j_h}, \phi_{i_h,j_h}\rangle \in S$, for all $h, 1 \leq h \leq m$, holds by induction on $h$.

Basis case ($h = 1$): In this case, there must exist a single walk $q_{1,0} \rightsquigarrow_{P_1} q_{1,k} = \xi_1$ in $M_1$ and a walk $q_{2,0} \rightsquigarrow_{P_2} q_{2,l} = \zeta_1$ in $M_2$ such that $\xi_1 \simeq \zeta_1$. Let the walk $\xi_1$ consist of the paths $\langle \alpha_1^1, \alpha_1^2, \ldots, \alpha_1^t \rangle$ from $P_1$; similarly, let $\zeta_1$ consist of the paths $\langle \beta_1^1, \beta_1^2, \ldots, \beta_1^t \rangle$ from $P_2$. Thus, we have $\sigma_{1,k} = s_{\alpha_1^t}(s_{\alpha_1^{t-1}} \ldots (s_{\alpha_1^1}(\sigma_{1,0})) \ldots)$ and $\sigma_{2,l} = s_{\beta_1^t}(s_{\beta_1^{t-1}} \ldots (s_{\beta_1^1}(\sigma_{1,0})) \ldots)$ from Definition 14. So, in clause 2 of Definition 11, let $q_{1,i} = q_{1,0}, \sigma_{1,i} = \sigma_{1,0}, \sigma_{1,k} = s_{\alpha_1^t}(s_{\alpha_1^{t-1}} \ldots (s_{\alpha_1^1}(\sigma_{1,0})) \ldots)$ so that $\sigma_{1,k} \in q_{1,k}(\overline{v}), \eta_1 = \xi_1, q_{2,j} = q_{2,0}, \sigma_{2,j} = \sigma_{2,0}$. We notice that the antecedents of clause 2 in Definition 11 hold; specifically, the antecedent $\langle q_{1,0}, \sigma_{1,0} \rangle \rightsquigarrow^{\eta_1} \langle q_{1,k}, \sigma_{1,k} \rangle$ holds by the right hand side (rhs) of clause 2 in Definition 18; the antecedent $\phi_{0,0}(\sigma_{1,0}, \sigma_{2,0})$ holds since $\phi_{0,0}$ is identically *true*; $\langle q_{1,0}, q_{2,0}, \phi_{0,0} \rangle \in S$ by clause 1 in Definition 11. Hence, the consequents of clause 2 in Definition 11 hold with $\sigma_{2,l} = s_{\beta_1^t}(s_{\beta_1^{t-1}} \ldots (s_{\beta_1^1}(\sigma_{1,0})) \ldots), \eta_2 = \zeta_1$; so from the fourth consequent, we have $\langle q_{1,k}, q_{2,l}, \phi_{k,l} \rangle \in S$.
Induction hypothesis: Let $\langle q_{1,k}, q_{2,l}, \phi_{k,l} \rangle \in S$ hold whenever the states $q_{1,k}$ in $Q_1$ and $q_{2,l}$ in $Q_2$ are reachable from $q_{1,0}$ and $q_{2,0}$, respectively, through sequences of length $n$ of pairwise equivalent walks from $P_1$ and $P_2$.
Induction step: Let the state $q_{1,k}$ be reachable from $q_{1,0}$ through the sequence $q_{1,0} \rightsquigarrow^{\xi_1} q_{1,i_1} \rightsquigarrow^{\xi_2} \cdots \rightsquigarrow^{\xi_n} q_{1,i_n} \rightsquigarrow^{\xi_{n+1}} q_{1,k}$ in $M_1$ and the state $q_{2,l}$ be reachable from $q_{2,0}$ through the sequence $q_{2,0} \rightsquigarrow^{\zeta_1} q_{2,j_1} \rightsquigarrow^{\zeta_2} \cdots \rightsquigarrow^{\zeta_n} q_{2,j_n} \rightsquigarrow^{\zeta_{n+1}} q_{2,l}$ in $M_2$ such that $\xi_h \simeq \zeta_h, 1 \leq h \leq n+1$. Now $\langle q_{1,i_n}, q_{2,j_n}, \phi_{i_n,j_n} \rangle \in S$ by the induction hypothesis. Let $\xi_{n+1}$ consist of the paths $\langle \alpha_{n+1}^1, \alpha_{n+1}^2, \ldots, \alpha_{n+1}^{t'} \rangle$ from $P_1$; similarly, let $\zeta_{n+1}$ consist of the paths $\langle \beta_{n+1}^1, \beta_{n+1}^2, \ldots, \beta_{n+1}^{t'} \rangle$ from $P_2$. So, in clause 2 in Definition 11, let $q_{1,i} = q_{1,i_n}, \sigma_{1,i} = \sigma_{1,i_n}, \sigma_{1,k} = s_{\alpha_{n+1}^{t'}}(s_{\alpha_{n+1}^{t'-1}} \ldots (s_{\alpha_{n+1}^1}(\sigma_{1,i_n})) \ldots)$ so that $\sigma_{1,k} = q_{1,k}(\overline{v}), \eta_1 = \xi_{n+1}, q_{2,j} = q_{2,j_n}, \sigma_{2,j} = \sigma_{2,j_n}$. We notice that the antecedent $\langle q_{1,i_n}, \sigma_{1,i_n} \rangle \rightsquigarrow^{\eta_1} \langle q_{1,k}, \sigma_{1,k} \rangle$ holds by the rhs of clause 2 in Definition 18; the antecedents $\phi_{i_n,j_n}(\sigma_{1,i_n}, \sigma_{2,j_n})$ and $\langle q_{1,i_n}, q_{2,j_n}, \phi_{i_n,j_n} \rangle \in S$ in clause 2 in Definition 18 hold by the induction hypothesis. Hence, the consequents of clause 2 in Definition 11 hold with $\sigma_{2,l} = s_{\beta_{n+1}^{t'}}(s_{\beta_{n+1}^{t'-1}} \ldots (s_{\beta_{n+1}^1}(\sigma_{2,j_n})) \ldots), \eta_2 = \zeta_{n+1}$; so from the fourth consequent, we have $\langle q_{1,k}, q_{2,l}, \phi_{k,l} \rangle \in S$. ■

The following theorem captures the fact that symbolic value propagation based equivalence checking leads to a simulation relation between two FSMDs.

**Theorem 9.** *If a symbolic value propagation based equivalence checker declares an FSMD $M_1$ to be contained in another FSMD $M_2$ for a path cover $P_1$ of $M_1$, then there is a simulation relation between $M_1$ and $M_2$ corresponding to the state correspondence relation $\delta$ produced by the equivalence checker. Symbolically,*

**Algorithm 7** deriveSimRelVR2 (FSMD $M_1$, FSMD $M_2$, Set $\delta$, Set $\hat{\delta}$, Set $\Upsilon$)

```
Inputs: Two FSMDs $M_1$ and $M_2$,
the sets $\delta$ and $\hat{\delta}$ of corresponding state pairs and corresponding loop state pairs,
the set of $\Upsilon_{k,l}$'s, $\Upsilon_{k,l} = \{\langle \overline{\vartheta_{1,k}}, \overline{\vartheta_{2,l}} \rangle \mid \langle q_{1,k}, q_{2,l} \rangle \in \delta_c\}$, where $\overline{\vartheta_{i,j}} = \langle C_j, \langle e_1, \ldots, e_{|V_0 \cup V_1|} \rangle \rangle$ represents a propagated vector at state $q_{i,j}$;
the last three arguments are obtained from $M_1$ and $M_2$ by the symbolic value propagation based equivalence checker.
Outputs: Relation $\mathcal{V}_2 = \{\langle q_{1,i}, q_{2,j}, \phi_{i,j} \rangle \mid \langle q_{1,i}, q_{2,j} \rangle \in \delta\}$,
         relation $\hat{\mathcal{V}} = \{\langle q_{1,i}, q_{2,j}, \phi_{i,j} \rangle \mid \langle q_{1,i}, q_{2,j} \rangle \in \hat{\delta}\}$.
1: Let the relations $\mathcal{V}_2$ and $\hat{\mathcal{V}}$ be empty.
2: Perform live variable analyses on $M_1$ and $M_2$ and compute the set of live variables for each of the states that appears as a member of the state pairs in $\delta$ and $\hat{\delta}$.
3: For the pair $\langle q_{1,0}, q_{2,0} \rangle$ in $\delta$, let $\phi_{0,0}$ be $true$ and $\mathcal{V}_2 \leftarrow \mathcal{V}_2 \cup \{\langle q_{1,0}, q_{2,0}, \phi_{0,0} \rangle\}$.
4: Rename each $v_j \in V_i$ as $v_{i,j}, i \in \{1,2\}$.
5: For each of the other state pairs $\langle q_{1,i}, q_{2,j} \rangle$ in $\delta$, let $\phi_{i,j}$ be $v_{1,k_1} = v_{2,k_1} \wedge \ldots \wedge v_{1,k_n} = v_{2,k_n}$, where $v_{k_1}, \ldots, v_{k_n}$ are the common variables live at $q_{1,i}$ and $q_{2,j}$; $\mathcal{V}_2 \leftarrow \mathcal{V}_2 \cup \{\langle q_{1,i}, q_{2,j}, \phi_{i,j} \rangle\}$.
6: Rename each $v_j \in V_i$ as $v_{i,j}, i \in \{1,2\}$ in the pairs in $\Upsilon_{k,l}$'s, with $i = 1(2)$ for the first (second) member of a pair.
7: *For each state pair $\langle q_{1,i}, q_{2,j} \rangle$ in $\hat{\delta}$ and corresponding $\Upsilon_{i,j}$,
     $\phi_{i,j} \leftarrow \exists v'_{1,h_1}, .., v'_{1,h_x} \exists v'_{2,h_1}, .., v'_{2,h_y}$
     $[\bigwedge_{g=1}^{m} v_{1,h_g} = v_{2,h_g} \wedge \bigwedge_{g=m+1}^{x} v_{1,h_g} = e_{1,h_g}(v'_{1,h_1}, .., v'_{1,h_x}) \wedge \bigwedge_{g=m+1}^{y} v_{2,h_g} = e_{2,h_g}(v'_{2,h_1}, .., v'_{2,h_y})]$,
   where $v_{h_1}, \ldots, v_{h_m}$ are the common variables live at $q_{1,i}$ and $q_{2,j}$ that assume identical values in both the FSMDs, $v_{1,h_{m+1}}, \ldots, v_{1,h_x}$ are the live common variables and uncommon variables that assume the symbolic expression values $e_{1,h_{m+1}}, \ldots, e_{1,h_x}$ at $q_{1,i}$ in FSMD $M_1$ and $v_{2,h_{x+1}}, \ldots, v_{2,h_y}$ are the live common variables and uncommon variables that assume the symbolic values $e_{2,h_{m+1}}, \ldots, e_{2,h_y}$ at $q_{2,j}$ in FSMD $M_2$ with mismatch in the common variables among them;   $\hat{\mathcal{V}} \leftarrow \hat{\mathcal{V}} \cup \{\langle q_{1,i}, q_{2,j}, \phi_{i,j} \rangle\}$.
8: return $\mathcal{V}_2$, $\hat{\mathcal{V}}$.
```

*Note that the variables $v_{1,h_1}, .., v_{1,h_x}, v_{2,h_1}, .., v_{2,h_y}$ in $\phi_{i,j}$ are all free, i.e., implicitly universally quantified when validity of $\phi_{i,j}$ is concerned.

$\forall \langle q_{1,i}, q_{2,j} \rangle \in Q_1 \times Q_2,\ \delta(q_{1,i}, q_{2,j}) \Rightarrow \exists \phi_{i,j}\ S(q_{1,i}, q_{2,j}, \phi_{i,j})$.

*Proof:* We actually prove that $\forall \langle q_{1,i}, q_{2,j} \rangle \in Q_1 \times Q_2,\ \delta(q_{1,i}, q_{2,j}) \Rightarrow \exists \phi_{i,j}\ \langle q_{1,i}, q_{2,j}, \phi_{i,j} \rangle \in \mathcal{V}'$, the type-II verification relation, and then apply Theorem 8 to infer that $\langle q_{1,i}, q_{2,j}, \phi_{i,j} \rangle \in S$.
Construction of $\phi_{i,j}$'s and a verification relation $\mathcal{V}_2$: Algorithm 7 shows the steps to obtain a simulation relation $\mathcal{V}_2$ from two FSMDs and the outputs of a symbolic value propagation based equivalence checker for those FSMDs; the algorithm also outputs a relation $\hat{\mathcal{V}}$ which is used subsequently for devising a checking algorithm for our bisimulation relation. Note that $\hat{\mathcal{V}}$ also contains tuples of the form $\langle q'_{1,i}, q'_{2,j}, \phi'_{i,j} \rangle$, similar to those of $\mathcal{V}_2$; however, while for a member $\langle q_{1,i}, q_{2,j}, \phi_{i,j} \rangle \in \mathcal{V}_2$, $\langle q_{1,i}, q_{2,j} \rangle \in \delta$ and $\phi_{i,j}$ involves equalities of the respective common live variables at $q_{1,i}$ and $q_{2,j}$, for a member $\langle q'_{1,i}, q'_{2,j}, \phi'_{i,j} \rangle \in \hat{\mathcal{V}}$, $\langle q'_{1,i}, q'_{2,j} \rangle$ belongs to $\hat{\delta}$ and $\phi'_{i,j}$ comprises equalities of two FSMD variables, if they match, and equalities of the mismatched variables with their respective symbolic values, otherwise.

Now, we prove that the verification relation $\mathcal{V}_2$ constructed in Algorithm 7 is indeed a type-II verification relation, i.e., the relation $\mathcal{V}_2$ conforms with Definition 18. Consider any $\langle q_{1,k}, q_{2,l}, \phi_{k,l} \rangle \in \mathcal{V}_2$. The triple $\langle q_{1,k}, q_{2,l}, \phi_{k,l} \rangle$ must have been put in $\mathcal{V}_2$ either in step 3 (case 1) or in step 5 (case 2).
For case 1, $q_{1,k} = q_{1,0}$, $q_{2,l} = q_{2,0}$ and $\phi_{k,l} =\ true$. From clause 1 of Definition 18, $\langle q_{1,0}, q_{2,0},\ true \rangle \in \mathcal{V}'$.
For case 2, from step 5 of Algorithm 7, it follows that $\langle q_{1,k}, q_{2,l} \rangle \in \delta$ and $L_{1,k}(\overline{v_{1c}}) = L_{2,l}(\overline{v_{2c}})$. From Definition 16 of $\delta$, it follows that there exists a sequence of configurations of $M_1$ of the form $\langle q_{1,0}, \sigma_{1,0} \rangle \rightsquigarrow^{\xi_1} \langle q_{1,i_1}, \sigma_{1,i_1} \rangle \rightsquigarrow^{\xi_2} \langle q_{1,i_2}, \sigma_{1,i_2} \rangle \rightsquigarrow^{\xi_3} \cdots \rightsquigarrow^{\xi_n} \langle q_{1,i_n} = q_{1,k}, \sigma_{1,i_n} = \sigma_{1,k} \rangle$, where $\xi_1, \xi_2, \ldots, \xi_n \in W_1$ comprising paths from $P_1$ and a sequence of configurations of $M_2$ of the form $\langle q_{2,0}, \sigma_{2,0} \rangle \rightsquigarrow^{\zeta_1} \langle q_{2,j_1}, \sigma_{2,j_1} \rangle \rightsquigarrow^{\zeta_2} \langle q_{2,j_2}, \sigma_{2,j_2} \rangle \rightsquigarrow^{\zeta_3} \cdots \rightsquigarrow^{\zeta_n} \langle q_{2,j_n} = q_{2,l}, \sigma_{2,j_n} = \sigma_{2,l} \rangle$, where $\zeta_1, \zeta_2, \ldots, \zeta_n \in W_2$ comprising paths from $P_2$, such that $\langle q_{1,i_h}, q_{2,j_h} \rangle \in \delta, 1 \le h \le n$. Hence, from step 5 of Algorithm 7, $\langle q_{1,i_h}, q_{2,j_h}, \phi_{i_h,j_h} \rangle \in \mathcal{V}_2, 1 \le h \le n$. We show that $\langle q_{1,i_h}, q_{2,j_h}, \phi_{i_h,j_h} \rangle \in \mathcal{V}'$, for all $h, 1 \le h \le n$, by induction on $h$.
*Basis ($h = 1$):* Let us consider the rhs of the biconditional in clause 2 of Definition 18 with $q_{1,k} = q_{1,i_1}$ and $q_{2,l} = q_{2,j_1}$. Let $q_{1,i} = q_{1,0}$; the first conjunct $q_{1,i} \rightsquigarrow_{P_1} q_{1,i_1}$ holds because $q_{1,0} \rightsquigarrow_{P_1} q_{1,k} = \xi_1$. The second conjunct holds with $q_{1,i} = q_{1,0}, q_{2,j} = q_{2,0}$, $q_{2,0} \rightsquigarrow_{P_2} (q_{2,k} = q_{2,j_1}) = \zeta_1$ and $\xi_1 \simeq \zeta_1$ and $\langle q_{1,0}, q_{2,0}, \phi_{0,0} \rangle \in \mathcal{V}'$ from clause 1 of

Definition 18. Hence, the lhs of the biconditional in clause 2 of Definition 18 holds yielding $\langle q_{1,i_1}, q_{2,j_1}, \phi_{i_1,j_1} \rangle \in \mathcal{V}'$.
*Induction hypothesis:* Let $\langle q_{1,i_h}, q_{2,j_h}, \phi_{i_h,j_h} \rangle \in \mathcal{V}', 1 \leq h \leq m < n$.
*Induction step ($h = m+1$):* Let $q_{1,k} = q_{1,i_{m+1}}, q_{2,l} = q_{2,j_{m+1}}$ in clause 2 of Definition 18. From the rhs of the biconditional in clause 2, let $q_{1,i} = q_{1,i_m}$; the first conjunct in the rhs namely, $q_{1,i} \rightsquigarrow_{P_1} q_{1,i_{m+1}} = \xi_{m+1}$, holds with $q_{1,i} = q_{1,i_m}$. In the second conjunct of the rhs, with $q_{1,i} = q_{1,i_m}$, the antecedent $q_{1,i} \rightsquigarrow_{P_1} q_{1,k} = q_{1,i_m} \rightsquigarrow_{P_1} q_{1,i_{m+1}} = \xi_{m+1}$ holds. In the consequent of this conjunct, let $q_{2,j} = q_{2,j_m}$; we find $q_{2,j} \rightsquigarrow_{P_2} q_{2,l} = q_{2,j_m} \rightsquigarrow_{P_2} q_{2,j_{m+1}} = \zeta_{m+1}$ holds; $\xi_{m+1} \simeq \zeta_{m+1}$ holds and $\langle q_{1,i}, q_{2,j}, \phi_{i,j} \rangle = \langle q_{1,i_m}, q_{2,j_m}, \phi_{i_m,j_m} \rangle \in \mathcal{V}'$ by induction hypothesis. So, the rhs of the biconditional holds. Hence, the lhs holds yielding $\langle q_{1,i_{m+1}}, q_{2,j_{m+1}}, \phi_{i_{m+1},j_{m+1}} \rangle \in \mathcal{V}'$. ■

If the symbolic value propagation based equivalence checker also finds that $M_2 \sqsubseteq M_1$, then it does so without changing $\delta$ and $\hat{\delta}$. Hence, the simulation relation $\mathcal{V}_2$ obtained from Algorithm 7 is a bisimulation relation because it conforms with Definition 12.

Similar to what has been mentioned in Section 4.3, Algorithm 7 also produces a bisimulation relation comprising triples for only the corresponding states. This, however, does not impair generality because if one is interested in enhancing the bisimulation relation by incorporating triples for some control states other than corresponding states, one may easily do so by applying Dijkstra's weakest precondition computation starting from the corresponding states appearing immediately next to one's states of choice in the control flow graph of FSMD; however, if one's states of choice lie within a loop, then one should start from the next corresponding loop state pairs $\langle q_{1,m}, q_{2,n} \rangle$, say, belonging to $\hat{\delta}$ and compute weakest precondition with respect to $\phi_{m,n}$.

## 4.5 Conclusion

Both bisimulation relation based methods and path based equivalence checking approaches are prevalent in the literature on translation validation of programs. The basic methodologies of these two approaches differ; the (conventional) bisimulation relation based approach tries to construct a relation that serves as a witness of the two programs being symbolically executed in an equivalent manner, whereas, the path based approach tries to obtain path covers in the two FSMDs such that each path in one is found to be equivalent with a path in the other and vice-versa. In this chapter, we relate these two (apparently different) approaches by explaining how bisimulation relations can be derived from the outputs of two types of path based equivalence checkers namely, a path extension based checker and a symbolic value propagation based checker. None of the bisimulation relation based approaches has been shown to tackle code motions across loops; therefore, the present work demonstrates, for the first time, that a bisimulation relation exists even under such a context when such code motions are valid. Developing a unified framework that encompasses all the benefits of these two approaches seems to be an interesting future work.

# Chapter 5

# Translation Validation of Code Motion Transformations in Array-Intensive Programs

## 5.1 Introduction

In Chapter 3, we have presented a symbolic value propagation based equivalence checking for the FSMD model. A significant deficiency of this method is its inability to handle an important class of programs, namely those involving arrays. The data flow analysis for array-intensive programs is notably more complex than those involving only scalars. To illustrate the fact, let us consider two sequential statements $a[i] \Leftarrow 10$ and $a[j] \Leftarrow 20$. Now consider the scenario where $i = j$ holds, in this case the second statement qualifies as an *overwrite*, whereas in the complement scenario of $i \neq j$, it does not. Unavailability of relevant information to resolve such relationships between index variables may result in an exponential number of case analyses. In addition, obtaining the condition of execution and the data transformation of a path by applying simple substitution as outlined by Dijkstra's weakest precondition computation may become more expensive in the presence of arrays; conditional clauses need to be associated depicting equality/non-equality of the index expressions of the array references in the predicate as it gets transformed through the array assignment statements in the path.

We first address the problem of deriving a succinct representation of expressions involving arrays so that the computation of the conditions and data transformations of paths can avoid case analysis. Towards this, we have borrowed the well-known McCarthy's *read* and *write* functions [120] (originally known as *access* and *change*, respectively) to represent assignment and conditional statements involving arrays that easily captures the sequence of transformations carried out on the elements of an array and also allows uniform substitution policy for both scalars and array variables. We then enhance the symbolic value propagation based method described in Chapter 3 to propagate the values assumed by the array variables and their corresponding index variables in some path to its subsequent paths; a special rule for the propagation of index variables is also incorporated.

The chapter is organized as follows. Section 5.2 introduces the finite state machine with datapath *having arrays* (FSMDA) model, which is an extension on the FSMD model equipped to efficiently handle arrays. The computation of the characteristic formula of a path is presented in Section 5.3. An advancement of the existing normalization technique to represent expressions involving arrays is given in Section 5.4. The overall verification method, illustrated with an example, can be found in Section 5.5. A theoretical analysis of the method is given in Section 5.6. Section 5.7 contains the experimental results along with a brief discussion on the current limitations of our method. The chapter is concluded in Section 5.8.

## 5.2 The FSMDA model

An FSMDA is formally defined as an ordered tuple $\langle Q, q_0, I, V, O, \tau : Q \times 2^S \rightarrow Q$, $h : Q \times 2^S \rightarrow U \rangle$, where $Q$ is the finite set of control states, $q_0$ is the reset state, $I$ is the set of input variables, $V$ is the set of storage variables, $O$ is the set of output variables, $\tau$ is the state transition function, $h$ is the update function of the output and the storage variables, $U$ represents a set of storage and output assignments of arithmetic expressions and the set $S$ represents a set of status signals as relations between two arithmetic expressions. The sets $I$, $V$ and $O$, unlike FSMD, are further partitioned into subsets ($I_s$, $I_a$), ($V_s$, $V_a$, $V_i$) and ($O_s$, $O_a$) respectively; suffix $s$ stands for *scalar*, $a$ for *array* and $i$ for *index*. Index variables are basically "scalar" variables that occur in some index expression of some array variable.

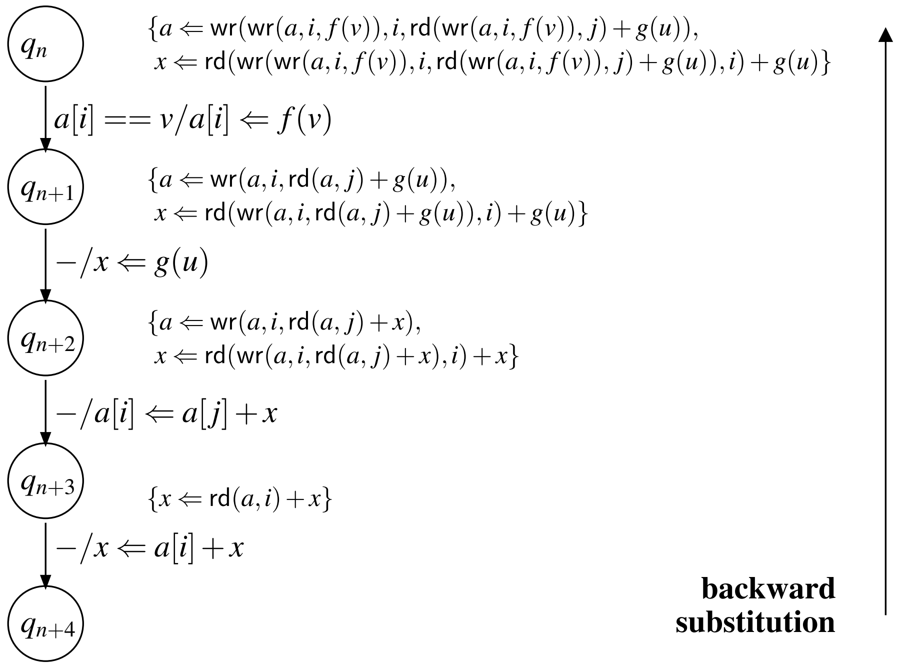

$q_n$
$\{a \Leftarrow \mathsf{wr}(\mathsf{wr}(a,i,f(v)),i,\mathsf{rd}(\mathsf{wr}(a,i,f(v)),j)+g(u)),$
$x \Leftarrow \mathsf{rd}(\mathsf{wr}(\mathsf{wr}(a,i,f(v)),i,\mathsf{rd}(\mathsf{wr}(a,i,f(v)),j)+g(u)),i)+g(u)\}$
$a[i] == v / a[i] \Leftarrow f(v)$
$q_{n+1}$
$\{a \Leftarrow \mathsf{wr}(a,i,\mathsf{rd}(a,j)+g(u)),$
$x \Leftarrow \mathsf{rd}(\mathsf{wr}(a,i,\mathsf{rd}(a,j)+g(u)),i)+g(u)\}$
$- / x \Leftarrow g(u)$
$q_{n+2}$
$\{a \Leftarrow \mathsf{wr}(a,i,\mathsf{rd}(a,j)+x),$
$x \Leftarrow \mathsf{rd}(\mathsf{wr}(a,i,\mathsf{rd}(a,j)+x),i)+x\}$
$- / a[i] \Leftarrow a[j]+x$
$q_{n+3}$
$\{x \Leftarrow \mathsf{rd}(a,i)+x\}$
$- / x \Leftarrow a[i]+x$
$q_{n+4}$
backward substitution

Figure 5.1: Computing characteristic tuple of a path.

## 5.3 Characteristic tuple of a path

An FSMDA, like an FSMD, can be fragmented into a set of paths by introducing cut-points in it to cut each loop; each path originates from a cut-point and ends in a cut-point. In this work, the reset state and all those states from which multiple outgoing transitions occur have been considered as the cut-points for an FSMDA. A path $\beta$ is characterized by a tuple $\langle R_\beta, s_\beta, \theta_\beta \rangle$, where $R_\beta$ is the *condition of execution*, $s_\beta$ is the *data transformation* of the path, and $\theta_\beta$ is the sequence of outputs produced by the path. The characteristic tuple captures that if $R_\beta$ is satisfied by $\bar{v}$, a vector of variables of $I \bigcup V$, at the beginning of $\beta$, then the path is traversed and after traversal the updated values of the variables of $V$ are given by $s_\beta(\bar{v})$ along with the output $\theta_\beta(\bar{v})$. Details about computation of the characteristic tuple can be found in [95].

In order to represent the assignment statements and the conditional statements involving arrays, we borrow the *read* (rd) and *write* (wr) functions introduced by McCarthy [120] because of the following reasons: (i) an elegant representation of the characteristic of a path is achieved with each array being depicted as a vector; (ii) the sequence of transformations of elements of arrays can be captured; and (iii) the substitution operations on scalar and array variables while finding the characteristic tuple of a path can be carried out identically. For example, the operation $a[i] \Leftarrow b[i]+z$ is represented as $\mathsf{wr}^{(3)}(a,i,\mathsf{rd}^{(2)}(b,i)+z)$. Originally, the functions have been proposed in [120] for only one-dimensional arrays, where rd has arity two and wr has arity three. We generalize the functions for $n$-dimensional arrays with their arities suitably modified (to accommodate the $n$ index expressions). However, for brevity, we shall omit the arities of these functions whenever it is clear from the context. Henceforth, the data transformations in FSMDAs will be represented using McCarthy's operations.

Figure 5.1 illustrates the process of computation of the characteristic of a path, say $\gamma$, in an FSMDA. Here $R_\gamma \equiv (\mathsf{rd}(a,i) == v)$, $s_\gamma = \{a \Leftarrow \mathsf{wr}(\mathsf{wr}(a,i,f(v)),i,\mathsf{rd}(\mathsf{wr}(a,i,f(v)),j)+g(u))$, $x \Leftarrow \mathsf{rd}(\mathsf{wr}(\mathsf{wr}(a,i,f(v)),i,\ \mathsf{rd}(\mathsf{wr}(a,i,f(v)),j)+g(u)),i)+g(u)\}$, and $\theta_\gamma$ is empty. The data transformation $s_\gamma$ can be computed by using the backward substitution method (also known as weakest precondition computation) as shown in Figure 5.1.

## 5.4 Normalization of expressions involving arrays

To represent arithmetic and logical expressions the normalization technique described in [141] has traditionally been used [21, 25, 90, 95, 110]. The problem of determining equivalence of two arbitrary arithmetic expressions over integers is undecidable. Normalization of arithmetic expressions is the first step whereby their structural similarity is targeted; many equivalent formulae become syntactically identical in the process.

On application of the grammar rules in [141], it is possible to convert any arithmetic expression involving integer variables and constants into its normalized form. The set of grammar rules has been updated by addition of the last production rule in the subset 3 and introduction of the rule 5, as given below, to accommodate arrays.

**Updated grammar:**

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

2) $T \rightarrow T*P \big| c_t$, where $c_t$ is an integer.

3) $P \rightarrow \mathrm{abs}(S) \big| (S)\,\mathrm{mod}(C_d) \big| S \div C_d | v | c_m | A$, where $v \in I_s \bigcup V_s \bigcup V_i$, and $c_m$ is an integer.

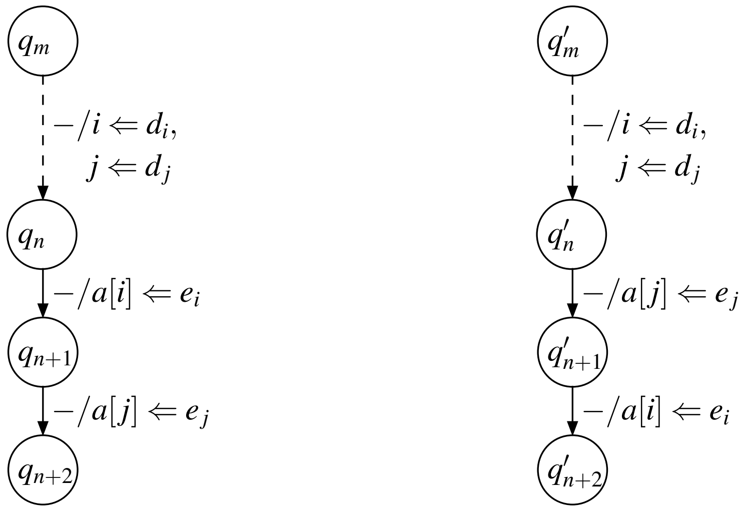
4) $C_d \rightarrow S \div C_d \big| (S)\, \mathrm{mod}(C_d) \big| S$.

5) $A \rightarrow \mathsf{wr}^{(k+2)}(v', S_1, \ldots, S_k, S) \big| \mathsf{rd}^{(k+1)}(v', S'_1, \ldots, S'_k)$, where $v' \in I_a \bigcup V_a$, and $S_1, \ldots, S_k, S'_1, \ldots, S'_k$ are of type $S$ (sum) involving variables in $V_i$.

In the above grammar, the non-terminals $S$, $T$, $P$ stand for (normalized) sums, terms and primaries, respectively, $A$ is an array primary, and $C_d$ is a divisor primary. The terminals are the variables belonging to $I \bigcup V$, the interpreted function constants abs, mod and $\div$ and the user defined uninterpreted function constants $f$. In addition to the syntactic structure, all expressions are ordered as follows: any normalized sum is arranged by lexicographic ordering of its constituent subexpressions from the bottom-most level, i.e., from the level of simple primaries assuming an ordering over the set of variables $I \bigcup V$; among the function terminals, abs $\prec \div \prec$ mod $\prec$ uninterpreted function constants. As such, all function primaries, including those involving the uninterpreted ones, are ordered in a term in an ascending order of their arities. Similarly, array primaries in a term are ordered in an ascending order of their dimensions. Primaries involving same array names are ordered in a term according to the order of their subscript expressions. Thus, a term of the form $\mathsf{rd}^{(2)}(a, s_6) * \mathsf{rd}^{(3)}(b, s_3, s_4) * \mathsf{rd}^{(3)}(b, s_2, s_5) * \mathsf{rd}^{(2)}(c, s_1)$ is ordered as $\mathsf{rd}^{(2)}(a, s_6) * \mathsf{rd}^{(2)}(c, s_1) * \mathsf{rd}^{(3)}(b, s_2, s_5) * \mathsf{rd}^{(3)}(b, s_3, s_4)$, where $a \prec c$ and the normalized subscript expressions (sums) have an ordering $s_2 \prec s_3$; the ordering of the remaining subscript expressions do not play any role in this case.

## 5.5 Equivalence checking of FSMDAs

The symbolic value propagation based equivalence checking method for FSMDs given in Chapter 3 is adapted for FSMDAs with a few additional steps. Basically, the method of symbolic value propagation consists in propagating the mismatched variable values (as *propagated vectors*) over a path to the subsequent paths until the values match or the final path segments are accounted for without finding a match. During the course of equivalence checking of two behaviours, two paths, say $\alpha$ and $\beta$, (one from each behaviour) are compared with respect to their corresponding propagated vectors for finding path equivalence. In case the computed characteristic tuples of the two paths match, they are declared as *unconditionally equivalent (U-equivalent)* (represented as



**(a)** $M_0$ **(b)** $M_1$

Figure 5.2: Propagation of index variables' values.

$\alpha \simeq \beta$); if some mismatch is detected, then they are declared as *conditionally equivalent (C-equivalent)* (represented as $\alpha \simeq_c \beta$) provided all the paths emanating from the final states of $\alpha$ and $\beta$ lead to some U-equivalent paths. In this work, the values of scalar and array variables are propagated to subsequent paths on encountering mismatches. However, the values of the index variables are propagated under all circumstances, irrespective of whether their values match in the corresponding paths or not, provided they are live (i.e., used subsequently prior to further definition). Therefore, unlike Chapter 3 where the propagated vectors are reset to their identity values upon finding a pair of U-equivalent paths, we continue to propagate the values to resolve subsequent (mis)matches for the variables. The following example underlines the justification.

**Example 7.** Let us consider the partial FSMDAs $M_1$ and $M_2$ given in Figure 5.2. Suppose $q_n$ and $q'_n$ are corresponding states, and we intend to find an equivalent path for $q_n \twoheadrightarrow q_{n+2}$, say $\alpha$, in $M_1$. When we compare the path $\alpha$ with the path $q'_n \twoheadrightarrow q'_{n+2}$, say $\beta$, we find that their data transformations differ; specifically, $s_\alpha = \{a \Leftarrow \mathsf{wr}(\mathsf{wr}(a,i,e_i),j,e_j)\}$ and $s_\beta = \{a \Leftarrow \mathsf{wr}(\mathsf{wr}(a,j,e_j),i,e_i)\}$. For the paths to be equivalent, one of the following cases must hold: (i) $e_i = e_j$ and $i = j$, (ii) $i \neq j$. In either case, the relation between $i$ and $j$ has to be ascertained. It is possible to infer whether $i = j$ or $i \neq j$ holds if their values at $q_n$ and $q'_n$, i.e. $d_i$ and $d_j$ respectively, are made available (by symbolic value propagation) at the states $q_n$ and $q'_n$, even if individually the $i$-values and the $j$-values are identical over the previous respective path segments, $q_m \twoheadrightarrow q_n$ and $q'_m \twoheadrightarrow q'_n$. Without such symbolic value propagation of the index variables, the equivalence between the array transformations will call for case analysis for each such nesting. ■

The undermentioned axioms [84] together with the propagated values of the index variables at the start states of the paths are used for resolving equivalence of array transformations over the paths.

$$\begin{aligned} i \neq j \supset \mathsf{rd}(\mathsf{wr}(\mathsf{wr}(a,i,e_1),j,e_2),i) = e_1 \ \wedge \\ \mathsf{rd}(\mathsf{wr}(\mathsf{wr}(a,i,e_1),j,e_2),j) = e_2 \end{aligned} \tag{5.1}$$

$$\begin{aligned} i = j \supset \mathsf{rd}(\mathsf{wr}(\mathsf{wr}(a,i,e_1),j,e_2),i) = e_2 \ \wedge \\ \mathsf{rd}(\mathsf{wr}(\mathsf{wr}(a,i,e_1),j,e_2),j) = e_2 \end{aligned} \tag{5.2}$$

In cases where $s_\alpha = \{a \Leftarrow \mathsf{wr}(\mathsf{wr}(a,i,e_1),j,e_2)\}$, $s_\beta = \{a \Leftarrow \mathsf{wr}(\mathsf{wr}(a,j,e_2),i,e_1)\}$ and $i \neq j$ hold, the data transformations $s_\alpha$ and $s_\beta$ are deemed equivalent by axiom (5.1), and where $s_\alpha = \{a \Leftarrow \mathsf{wr}(\mathsf{wr}(a,i,e_1),j,e_2)\}$, $s_\beta = \{a \Leftarrow \mathsf{wr}(a,j,e_2)\}$ and $i = j$ hold, $s_\alpha$ and $s_\beta$ are deemed equivalent by axiom (5.2).

The example given below illustrates the equivalence checking method of FSMDAs. The function equivalenceChecker (Algorithm 8) is then presented which describes the overall verification procedure in brief.

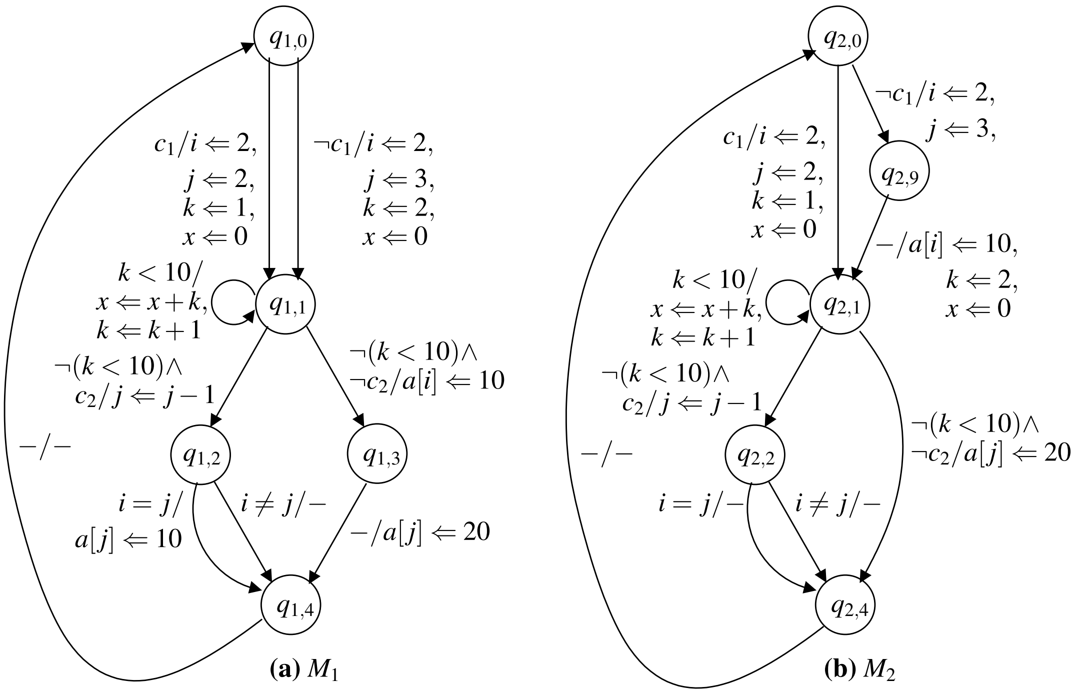
**Example 8.** Figure 5.3 shows two FSMDAs $M_1$ and $M_2$ having $q_{1,0}$ and $q_{2,0}$ as the reset states, respectively. Initially, we start from the reset states and for each path in $M_1$ we try to find a U-equivalent or C-equivalent path in $M_2$ following a depth-first traversal of the FSMDAs as shown below. The variable ordering for the vectors is $\langle i,j,a\rangle$.

Table 5.1 lists the steps involved; the letters U, C and N in the last column of the table stand for U-equivalent, C-equivalent and *not feasible*, respectively. Some of the steps, especially the *-marked ones, require a closer inspection as elaborated below in order.

*Step 1:* The paths $q_{1,0} \xrightarrow{c_1} q_{1,1}$ in $M_1$ and $q_{2,0} \xrightarrow{c_1} q_{2,1}$ in $M_2$ have the data transformation $\{i \Leftarrow 2, j \Leftarrow 2, k \Leftarrow 1, x \Leftarrow 0\}$ and the condition of execution $c_1$; hence they

Table 5.1: Computation of propagated vectors during equivalence checking of FSMDAs

| Step | $\alpha \in M_1$ | Initial Vector for $\alpha$ | $\beta \in M_2$, s.t. $\alpha \simeq_{[c]} \beta$ | Initial Vector for $\beta$ | Final Vector for $\alpha$ | Final Vector for $\beta$ | Decision |
|---|---|---|---|---|---|---|---|
| 1 | $q_{1,0} \xrightarrow{c_1} q_{1,1}$ | $\langle \top, \langle i,j,a,k,x \rangle \rangle$ | $q_{2,0} \xrightarrow{c_1} q_{2,1}$ | $\langle \top, \langle i,j,a,k,x \rangle \rangle$ | $\langle c_1, \langle 2,2,a,k,x \rangle \rangle$ | $\langle c_1, \langle 2,2,a,k,x \rangle \rangle$ | U |
| 2 | $q_{1,1} \xrightarrow{k<10} q_{1,1}$ | $\langle c_1, \langle 2,2,a,k,x \rangle \rangle$ | $q_{2,1} \xrightarrow{k<10} q_{2,1}$ | $\langle c_1, \langle 2,2,a,k,x \rangle \rangle$ | $\langle c_1, \langle 2,2,a,k,x \rangle \rangle$ | $\langle c_1, \langle 2,2,a,k,x \rangle \rangle$ | U |
| *3 | $q_{1,1} \xrightarrow{\neg(k<10) \wedge c_2} q_{1,2}$ | $\langle c_1, \langle 2,2,a,k,x \rangle \rangle$ | $q_{2,1} \xrightarrow{\neg(k<10) \wedge c_2} q_{2,2}$ | $\langle c_1, \langle 2,2,a,k,x \rangle \rangle$ | $\langle c_1 \wedge \neg(k<10) \wedge c_2,$ $\langle 2,1,a,k,x \rangle \rangle$ | $\langle c_1 \wedge \neg(k<10) \wedge c_2,$ $\langle 2,1,a,k,x \rangle \rangle$ | U |
| 4 | $q_{1,2} \xrightarrow{i=j} q_{1,0}$ | $\langle c_1 \wedge \neg(k<10) \wedge c_2,$ $\langle 2,1,a,k,x \rangle \rangle$ | – | – | – | – | N |
| *5 | $q_{1,2} \xrightarrow{i \neq j} q_{1,0}$ | $\langle c_1 \wedge \neg(k<10) \wedge c_2,$ $\langle 2,1,a,k,x \rangle \rangle$ | $q_{2,2} \xrightarrow{i \neq j} q_{2,0}$ | $\langle c_1 \wedge \neg(k<10) \wedge c_2,$ $\langle 2,1,a,k,x \rangle \rangle$ | $\langle c_1 \wedge \neg(k<10) \wedge c_2 \wedge i \neq j,$ $\langle 2,1,a,k,x \rangle \rangle$ | $\langle c_1 \wedge \neg(k<10) \wedge c_2 \wedge i \neq j,$ $\langle 2,1,a,k,x \rangle \rangle$ | U |
| *6 | $q_{1,1} \xrightarrow{\neg(k<10) \wedge \neg c_2} q_{1,0}$ | $\langle c_1, \langle 2,2,a,k,x \rangle \rangle$ | $q_{2,1} \xrightarrow{\neg(k<10) \wedge \neg c_2} q_{2,0}$ | $\langle c_1, \langle 2,2,a,k,x \rangle \rangle$ | $\langle c_1 \wedge \neg(k<10) \wedge \neg c_2,$ $\langle 2,2,\mathsf{wr}(a,2,20),k,x \rangle \rangle$ | $\langle c_1 \wedge \neg(k<10) \wedge \neg c_2,$ $\langle 2,2,\mathsf{wr}(a,2,20),k,x \rangle \rangle$ | U |
| 7 | $q_{1,0} \xrightarrow{\neg c_1} q_{1,1}$ | $\langle \top, \langle i,j,a,k,x \rangle \rangle$ | $q_{2,0} \xrightarrow{\neg c_1} q_{2,1}$ | $\langle \top, \langle i,j,a,k,x \rangle \rangle$ | $\langle \neg c_1, \langle 2,3,a,k,x \rangle \rangle$ | $\langle \neg c_1, \langle 2,3,\mathsf{wr}(a,2,10),k,x \rangle \rangle$ | C |
| *8 | $q_{1,1} \xrightarrow{k<10} q_{1,1}$ | $\langle \neg c_1, \langle 2,3,a,k,x \rangle \rangle$ | $q_{2,1} \xrightarrow{k<10} q_{2,1}$ | $\langle \neg c_1, \langle 2,3,\mathsf{wr}(a,2,10),k,x \rangle \rangle$ | $\langle \neg c_1, \langle 2,3,a,k,x \rangle \rangle$ | $\langle \neg c_1, \langle 2,3,\mathsf{wr}(a,2,10),k,x \rangle \rangle$ | C |
| 9 | $q_{1,1} \xrightarrow{\neg(k<10) \wedge c_2} q_{1,2}$ | $\langle \neg c_1, \langle 2,3,a,k,x \rangle \rangle$ | $q_{2,1} \xrightarrow{\neg(k<10) \wedge c_2} q_{2,2}$ | $\langle \neg c_1, \langle 2,3,\mathsf{wr}(a,2,10),k,x \rangle \rangle$ | $\langle \neg c_1 \wedge \neg(k<10) \wedge c_2,$ $\langle 2,2,a,k,x \rangle \rangle$ | $\langle \neg c_1 \wedge \neg(k<10) \wedge c_2,$ $\langle 2,2,\mathsf{wr}(a,2,10),k,x \rangle \rangle$ | C |
| 10 | $q_{1,2} \xrightarrow{i=j} q_{1,0}$ | $\langle \neg c_1 \wedge \neg(k<10) \wedge c_2,$ $\langle 2,2,a,k,x \rangle \rangle$ | $q_{2,2} \xrightarrow{i=j} q_{2,0}$ | $\langle \neg c_1 \wedge \neg(k<10) \wedge c_2,$ $\langle 2,2,\mathsf{wr}(a,2,10),k,x \rangle \rangle$ | $\langle \neg c_1 \wedge \neg(k<10) \wedge c_2 \wedge i=j,$ $\langle 2,2,\mathsf{wr}(a,2,10),k,x \rangle \rangle$ | $\langle \neg c_1 \wedge \neg(k<10) \wedge c_2 \wedge i=j,$ $\langle 2,2,\mathsf{wr}(a,2,10),k,x \rangle \rangle$ | U |
| 11 | $q_{1,2} \xrightarrow{i \neq j} q_{1,0}$ | $\langle \neg c_1 \wedge \neg(k<10) \wedge c_2,$ $\langle 2,2,a,k,x \rangle \rangle$ | – | – | – | – | N |
| 12 | $q_{1,1} \xrightarrow{\neg(k<10) \wedge \neg c_2} q_{1,0}$ | $\langle \neg c_1, \langle 2,3,a,k,x \rangle \rangle$ | $q_{2,1} \xrightarrow{\neg(k<10) \wedge \neg c_2} q_{2,0}$ | $\langle \neg c_1, \langle 2,3,\mathsf{wr}(a,2,10),k,x \rangle \rangle$ | $\langle \neg c_1 \wedge \neg(k<10) \wedge \neg c_2,$ $\langle 2,3,\mathsf{wr}(\mathsf{wr}(a,2,10),3,20),k,x \rangle \rangle$ | $\langle \neg c_1 \wedge \neg(k<10) \wedge \neg c_2,$ $\langle 2,3,\mathsf{wr}(\mathsf{wr}(a,2,10),3,20),k,x \rangle \rangle$ | U |


$q_{1,0}$
$c_1/i \Leftarrow 2,$ $j \Leftarrow 2,$ $k \Leftarrow 1,$ $x \Leftarrow 0$
$\neg c_1/i \Leftarrow 2,$ $j \Leftarrow 3,$ $k \Leftarrow 2,$ $x \Leftarrow 0$
$k < 10/$ $x \Leftarrow x+k,$ $k \Leftarrow k+1$
$q_{1,1}$
$\neg(k < 10) \wedge$ $c_2/j \Leftarrow j-1$
$\neg(k < 10) \wedge$ $\neg c_2/a[i] \Leftarrow 10$
$-/-$
$q_{1,2}$
$q_{1,3}$
$i = j/$ $a[j] \Leftarrow 10$
$i \neq j/-$
$-/a[j] \Leftarrow 20$
$q_{1,4}$
$q_{2,0}$
$\neg c_1/i \Leftarrow 2,$ $j \Leftarrow 3,$
$c_1/i \Leftarrow 2,$ $j \Leftarrow 2,$ $k \Leftarrow 1,$ $x \Leftarrow 0$
$q_{2,9}$
$-/a[i] \Leftarrow 10,$ $k \Leftarrow 2,$ $x \Leftarrow 0$
$k < 10/$ $x \Leftarrow x+k,$ $k \Leftarrow k+1$
$q_{2,1}$
$\neg(k < 10) \wedge$ $c_2/j \Leftarrow j-1$
$\neg(k < 10) \wedge$ $\neg c_2/a[j] \Leftarrow 20$
$-/-$
$q_{2,2}$
$i = j/-$
$i \neq j/-$
$q_{2,4}$

**(a)** $M_1$ **(b)** $M_2$

Figure 5.3: An example of equivalence checking of FSMDAs.

are identified to be U-equivalent; however, recursive search for U-equivalent or C-equivalent paths from the respective final states $q_{1,1}$ and $q_{2,1}$ is pursued with the propagated values of the index variables; this permits proper use of the equality antecedents of the axioms (1) or (2) to simplify data transformations of array elements in the subsequent path segments as explained in Example 7.

*Step 2:* The simple loops in the respective FSMDAs are now compared. Since their characteristic tuple match perfectly, these two loops are declared U-equivalent as well.

*Step 3:* When the paths $q_{1,1} \xrightarrow{\neg(k<10)\wedge c_2} q_{1,2}$ and $q_{2,1} \xrightarrow{\neg(k<10)\wedge c_2} q_{2,2}$ are compared, a similar decision, as in steps 1 and 2, is taken.

*Step 4:* The condition for the path $q_{1,2} \xrightarrow{i=j} q_{1,0}$ is not satisfied by the propagated values $i \Leftarrow 2, j \Leftarrow 1$; specifically, the original $R_\alpha \equiv (i = j)$ and under propagated vector $R'_\alpha \equiv (i = j)\{i/2, j/1\} \equiv (2 = 1) \equiv \mathit{false}$ – accordingly, search for a U-equivalent or C-equivalent path is abandoned through this path for the corresponding propagated vector.

*Step 5:* Upon considering the path $q_{1,2} \xrightarrow{i\neq j} q_{1,0}$, it is found to be U-equivalent with $q_{2,2} \xrightarrow{i\neq j} q_{2,0}$. Since the reset states are reached as the final states of the paths, further search for equivalent paths is not pursued.

*Step 6:* When the paths $q_{1,1} \xrightarrow{\neg(k<10)\wedge\neg c_2} q_{1,0}$ and $q_{2,1} \xrightarrow{\neg(k<10)\wedge\neg c_2} q_{2,0}$ are compared, their data transformations are found to be $\{i \Leftarrow 2, j \Leftarrow 2, a \Leftarrow \mathsf{wr}(\mathsf{wr}(a, 2, 10),$

$2,20)\}$ and $\{i \Leftarrow 2, j \Leftarrow 2, a \Leftarrow \mathsf{wr}(a,2,20)\}$, respectively. Since $i = j$, from axiom (5.2), the data transformation of $q_{1,1} \xrightarrow{\neg(k<10)\wedge\neg c_2} q_{1,0}$ can be rewritten as $\{i \Leftarrow 2, j \Leftarrow 2, a \Leftarrow \mathsf{wr}(a,2,20)\}$, making it equivalent to that of the path $q_{2,1} \xrightarrow{\neg(k<10)\wedge\neg c_2} q_{2,0}$. The values for the variable $a$ in both the paths match, and hence its symbolic value "a" is stored in the final propagated vector. The paths are declared U-equivalent. Since reset states have been reached, no further search from the respective final state is needed.

*Step 7:* The operation $a \Leftarrow \mathsf{wr}(a,i,10)$ in the path $q_{2,0} \xrightarrow{\neg c_1} q_{2,1}$ has no identical operation in the path $q_{1,0} \xrightarrow{\neg c_1} q_{1,1}$. Therefore, this is the first step where the transformed propagated vectors mismatch, and hence the two paths are declared to be *candidate* C-equivalent. The value for the array $a$ is propagated anticipating that a similar operation may eventually be discovered in $M_1$ making the subsequent paths U-equivalent. If all the subsequent paths eventually lead to some U-equivalent path, then such candidate C-equivalent paths are declared to be C-equivalent; otherwise they are adjudged to be inequivalent.

*Step 8:* The loops $q_{1,1} \xrightarrow{k<10} q_{1,1}$ and $q_{2,1} \xrightarrow{k<10} q_{2,1}$ with propagated vectors $\langle\neg c_1, \langle 2,3,a,k,x\rangle\rangle$ ($= \bar{\vartheta}$, say) and $\langle\neg c_1, \langle 2,3,\mathsf{wr}(a,2,10),k,x\rangle\rangle$ ($= \bar{\vartheta}'$), respectively, are compared; although the characteristic tuples of the loops match, the mismatch in the value of $a$ persists. Since the propagated vectors obtained during the entry and the exit of the loop for both the FSMDAs are identical, it indicates that code motion across loop may have occurred, i.e., if an assignment of the variable $a$ is found in $M_1$ which renders the values of $a$ in both the FSMDAs equivalent, then such code motion across loop will indeed be valid. So, these paths are declared as candidate C-equivalent and the search for U-equivalence is continued.

*Step 9:* Upon comparing the paths $q_{1,1} \xrightarrow{\neg(k<10)\wedge c_2} q_{1,2}$ and $q_{2,1} \xrightarrow{\neg(k<10)\wedge c_2} q_{2,2}$, the matching operation in $M_1$ is still not found, and the values for $a$ along with those of the index variables are further propagated.

*Step 10:* The matching operation is finally found in the path $q_{1,2} \xrightarrow{i=j} q_{1,0}$ making its data transformation same as that of $q_{2,2} \xrightarrow{i=j} q_{2,0}$, i.e. $\{i \Leftarrow 2, j \Leftarrow 2, a \Leftarrow \mathsf{wr}(a,2,10)\}$, confirming a valid code motion across loop and rendering these two paths U-equivalent.

*Step 11:* Similar situation as that of step 4 arises, and hence the path $q_{1,2} \xrightarrow{i\neq j} q_{1,0}$ is deemed to be not feasible for the corresponding propagated vector.

*Step 12:* Based on the propagated vectors $\bar{\vartheta}$ and $\bar{\vartheta}'$ at $q_{1,1}$ and $q_{2,1}$, the data transformations for both the paths $q_{1,1} \xrightarrow{\neg(k<10)\wedge\neg c_2} q_{1,0}$ and $q_{2,1} \xrightarrow{\neg(k<10)\wedge\neg c_2} q_{2,0}$ are found to be $\{i \Leftarrow 2, j \Leftarrow 3, a \Leftarrow \mathsf{wr}(\mathsf{wr}(a,2,10),3,20)\}$, and they are declared to be U-equivalent. The candidate C-equivalent paths identified in steps 7, 8 and 9 are also declared as actually C-equivalent.

In each of the steps 5, 6, 10 and 12, the reset states are revisited on traversing the path, which marks the end of a computation for the FSMDA. Note that any prevailing discrepancy in the vectors at this stage would indicate that the FSMDAs may not be equivalent. However, in this example, whenever the reset states are revisited, we find that the final propagated vectors in both the FSMDAs $M_1$ and $M_2$ are identical. Therefore, we can conclude $M_1 \sqsubseteq M_2$. Now, it is checked whether any path in $M_2$ remains whose equivalent path has not been found in $M_1$. Since no such path exists in $M_2$, the FSMDAs $M_1$ and $M_2$ are declared equivalent. ■

**Algorithm 8** equivalenceChecker (FSMDA $M_1$, FSMDA $M_2$)

```
1: Incorporate cut-points in $M_1$ and $M_2$ and compute the respective path covers, $P_1$ and $P_2$. // cf. Section 5.3
2: Normalize all the expressions. // cf. Section 5.4
3: Compute the characteristic tuple for each path. // cf. Section 5.3
4: Starting with the pair of reset states $\langle q_{1,0}, q_{2,0} \rangle$ choose paths from $P_1$ and $P_2$ in a depth-first manner. // cf. Section 5.5 for this step and all the subsequent steps
5: For every state pair comprising the final states of the chosen paths, propagate the values of the mismatched variables and the index variables.
6: If any mismatched value remains unresolved when any of the final paths of the computations (i.e., back to the reset states) of $M_1$ and $M_2$ are accounted for, declare *possibly* $M_1 \not\sqsubseteq M_2$; otherwise, declare $M_1 \sqsubseteq M_2$.
7: If any path of $P_2$ exists which does not pair with a path of $P_1$, then *possibly* $M_2 \not\sqsubseteq M_1$; otherwise, $M_2 \sqsubseteq M_1$ and hence declare $M_1 \equiv M_2$.
```

## 5.6 Correctness and complexity

### 5.6.1 Correctness

The overall equivalence checking method of FSMDAs primarily differs from that of FSMDs, as explained in Chapter 3, in two ways: (i) representation of array references using McCarthy's functions and (ii) propagation on index variable values in spite of a match. Since none of these modifications has any bearing on the correctness of the equivalence checking procedure for FSMDAs as compared to that of FSMDs, the proofs of soundness and termination for Algorithm 8 are explained here intuitively to avoid repetition.

**Soundness:** Once the cut-points have been introduced and the path covers have been constructed (step 1 in Algorithm 8), the equivalence checking method starts with the reset states and for each path in FSMDA $M_1$ searches for corresponding U-equivalent or C-equivalent path in the other FSMDA $M_2$ in a depth-first manner (steps 4 and 5). It declares the two FSMDAs equivalent only when the algorithm succeeds in finding U-equivalent or C-equivalent path pairs for every path in $M_1$ and $M_2$ (step 7), otherwise the FSMDAs are considered to be possibly non-equivalent (steps 6 and 7). Now, suppose that the algorithm is not sound, i.e., it declares two FSMDAs to be equivalent when there is a path, $\gamma$ say, in either of the FSMDAs containing an operation, $e$ say, whose match does not occur in the other FSMDA. The path $\gamma$ will obviously not have a U-equivalent path in the other FSMDA; moreover, it will neither have a C-equivalent path because, as already mentioned in the beginning of the previous section, two paths are declared to be C-equivalent only when all the paths emanating from their final states lead to some U-equivalent paths, i.e., the mismatched operation $e$ must have been compensated for eventually in the other FSMDA, which is a contradiction.

**Termination:** Since normalization of expressions and computing characteristic tuples of paths take finite number of steps and the number of paths in an FSMDA is finite, Algorithm 8 must terminate.

### 5.6.2 Complexity

Representation of array references by adopting McCarthy's functions does not change the complexity of normalization. On the other hand, propagation of variable values (index or otherwise) has to be done along all possible paths in the worst case, where the mismatches are not resolved until the last path in an FSMDA has been traversed; thus, the second difference on account of index variables leads to no ramification of the worst case time complexity of the equivalence checking method of FSMDAs

Table 5.2: Verification results of code transformations

| Benchmark | #arr | #op | #BB | #if | #loop | #path | Orig FSMDA | | Trans FSMDA | | Time |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | #state | #scalar | #state | #scalar | (ms) |
| ASSORT | 1 | 46 | 7 | 1 | 2 | 7 | 22 | 13 | 26 | 27 | 22 |
| BLOWFISH | 5 | 64 | 14 | 3 | 7 | 21 | 61 | 17 | 36 | 21 | 44 |
| FFT | 8 | 42 | 6 | 0 | 3 | 7 | 31 | 22 | 19 | 28 | 36 |
| GCD | 1 | 18 | 9 | 4 | 2 | 13 | 14 | 3 | 8 | 3 | 11 |
| GSR | 2 | 22 | 6 | 0 | 3 | 7 | 15 | 10 | 11 | 16 | 16 |
| LU–DECOMP | 2 | 56 | 18 | 2 | 8 | 21 | 47 | 8 | 44 | 20 | 21 |
| LU–PIVOT | 2 | 15 | 6 | 2 | 1 | 7 | 15 | 6 | 11 | 6 | 6 |
| LU–SOLVE | 4 | 34 | 10 | 0 | 5 | 11 | 28 | 4 | 22 | 14 | 12 |
| MINSORT | 1 | 22 | 6 | 1 | 2 | 7 | 20 | 7 | 11 | 7 | 7 |
| SB–BALANCE | 3 | 24 | 14 | 2 | 4 | 13 | 22 | 3 | 13 | 6 | 9 |
| SB–PC | 3 | 25 | 11 | 1 | 4 | 11 | 21 | 6 | 15 | 12 | 8 |
| SB–SAC | 3 | 21 | 11 | 1 | 4 | 11 | 20 | 5 | 14 | 8 | 8 |
| SVD | 4 | 279 | 82 | 17 | 36 | 107 | 250 | 25 | 220 | 77 | 93 |
| TR-LCOST | 4 | 74 | 25 | 4 | 9 | 27 | 56 | 14 | 46 | 31 | 35 |
| TR-NWEST | 5 | 72 | 25 | 4 | 9 | 27 | 54 | 13 | 47 | 29 | 31 |
| TR-VOGEL | 8 | 160 | 54 | 17 | 14 | 63 | 96 | 13 | 70 | 68 | 68 |
| WAVELET | 2 | 41 | 4 | 1 | 2 | 7 | 25 | 8 | 14 | 32 | 14 |
| ERRONEOUS | 1 | 22 | 5 | 1 | 2 | 7 | 16 | 6 | 15 | 10 | 6 |

in comparison to that of FSMDs. Therefore, the time complexity of Algorithm 8 is $O((k \cdot n + |V_1 \bigcup V_2|) \cdot 2^{\|F\|} \cdot n \cdot k^{(n-1)} \cdot (n-1)^{(n-1)})$ in the worst case, where $n$ is the number of states in the FSMDA, $k$ is the maximum number of parallel edges between any two states, $|V_1 \bigcup V_2|$ is the total number of variables in the two FSMDAs and $\|F\|$ is the maximum length of a normalized formula.

## 5.7 Experimental Results

Our symbolic value propagation based equivalence checker for FSMDAs has been implemented in C and the tool is available at http://cse.iitkgp.ac.in/~chitta/pubs/EquivalenceChecker_FSMDA.tar.gz along with the benchmarks, installation and usage guidelines. The benchmarks comprise behaviours which are computation intensive and which primarily involve arrays. ASSORT computes the degree assortativity of graphs, BLOWFISH implements the encryption algorithm of a block cipher, FFT performs fast Fourier transform, GCD finds the greatest common divisor of the members of an array, GSR implements Gauss-Seidel relaxation method, MINSORT is a sorting function, SVD computes singular value decomposition of a matrix, and WAVELET implements the Debaucles 4-coefficient wavelet filter. The benchmarks having “LU” as prefix are concerned with the LU decomposition of a matrix, which is a key step for several algorithms in linear algebra. The benchmarks having “SB” as prefix test cryptographic properties like balancedness, PC, and SAC for an S-box [142] whereas, those with “TR” are the various solutions to the transportation problem namely least cost method, northwest corner method and Vogel’s approximation method. ERRONEOUS is an example which reveals a bug in the implementation of copy propagation for array variables in the SPARK [64] tool that was first reported in [103]. These benchmarks are fed to the SPARK tool to obtain the transformed behaviours. New scalar variables, but no array variables, are introduced during the transformations. The proposed method can ascertain the equivalences in all the cases except the last one; in this case, it reports a possible non-equivalence and outputs a set of paths (each paired with an almost similar path) for which equivalence could not be found. The verification results are provided in Table 5.2.

The outputs of SPARK do not contain any “for” loops; the tool converts them into do-while loops. The tools of [154] and [88] which are existing equivalence checkers for array-intensive programs require the loops to be represented as “for” loops with their ranges clearly specified. Hence, the do-while loops in the outputs of the SPARK tool corresponding to for loops in the original specifications are manually restored to for loops and given along with the original programs to these tools for equivalence checking. These tools, however, still failed to establish the equivalence.

## 5.7.1 Current limitations

The method presented in this work has the following limitations.

(1) Arrays containing other arrays in their subscripts are not allowed. A possible remedy is to simply allow arrays to appear in $V_i$; however, our current implementation does not support it.

(2) It cannot handle loop transformations.

## 5.8 Conclusion

The literature on behavioural verification is rich with applications of the FSMD model involving only scalar variables. The model, and hence the verifiers built upon it, cannot handle arrays. To alleviate this limitation, the current chapter has made the following contributions. (i) A new model called FSMDA is introduced which is an extension of the FSMD model equipped to handle arrays. (ii) An improvisation of the normalization process [141] is suggested to represent arithmetic expressions involving arrays in normalized forms. (iii) The equivalence checker described in Chapter 3 is fitted with the FSMDA model to verify code transformations in array-intensive programs. The experiments carried out on benchmarks from various domains of embedded systems such as signal processing, data communication, hardware security, etc. attest the efficacy of the method. Enhancing the present method to overcome the current limitations stated above remains as our future goal.

# Chapter 6

# Translation Validation of Loop and Arithmetic Transformations in the Presence of Recurrences

## 6.1 Introduction

Loop transformations together with arithmetic transformations are applied extensively in the domain of multimedia and signal processing applications to obtain better performance in terms of energy, area and/or execution time [30, 79, 131, 165]. Loop transformations essentially involve partitioning/unifying the index spaces of arrays. Figure 6.1 shows two programs before and after application of loop fusion and tiling transformations. Specifically, two nested loops in the original program are fused into one and then the $8 \times 8$ index space of $\langle i, j \rangle$ is covered hierarchically along $4 \times 4$ tiles by the outer loop iterators $\langle l1, l2 \rangle$, each tile having $2 \times 2$ elements covered by the inner loop iterators $\langle l3, l4 \rangle$. Clearly, establishing equivalence/non-equivalence of the two programs shown in Figure 6.1 calls for an (elaborate) analysis of the index spaces of the involved arrays. The equivalence checking strategy involving the FSMDA model, as explained in the previous chapter, is not equipped to handle such reasoning over array index spaces; rather data dependence graph based analyses have been found to be suitable for verification of loop transformations.

```
for(i = 0; i <= 7; i++){
  for(j = 0; j <= 7; j++){
    S1 : A[i+1][j+1] = f(In[i][j]);
}}
for(i = 0; i <= 7; i++){
  for(j = 0; j <= 7; j++){
    S2 : b[i][j] = g(In[i][j]);
}}
```

(a) Original program.

```
for(l1 = 0; l1 <= 3; l1++){
  for(l2 = 0; l2 <= 3; l2++){
    for(l3 = 0; l3 <= 1; l3++){
      for(l4 = 0; l4 <= 1; l4++){
        i = 2*l1 + l3;
        j = 2*l2 + l4;
        S1 : a[i+1][j+1] = f(In[i][j]);
        S2 : b[i][j] = g(In[i][j]);
}}}}
```

(b) Transformed program.

Figure 6.1: Two programs before and after loop transformation.

An array data dependence graph (ADDG) based equivalence checking method has been proposed by Shashidhar et al. in [146] which is capable of verifying many loop transformations without requiring any supplementary information from the compiler. Another data dependence graph based method has been proposed by Verdoolaege et al. in [153, 154] that can additionally handle recurrences. However, none of these methods [146, 153, 154] can handle arithmetic transformations, such as distributive transformations, common sub-expression elimination, arithmetic expression simplification, constant (un)folding. The ADDG based method described in [86, 88] has been shown to handle loop and arithmetic transformations. However, since recurrences lead to cycles in the data-dependence graph of a program which make dependence analyses and simplifications (through closed-form representations) of the data transformations difficult, the method of [86, 88], which basically relies on such simplification procedures, fails in the presence of recurrences. This chapter describes a unified equivalence checking framework based on ADDGs to handle loop and arithmetic transformations along with recurrences. Specifically, the slice based ADDG equivalence checking framework [88] which handles loop and arithmetic transformations is extended so that recurrences are also handled.

The rest of the chapter is organized as follows. The class of input programs that is supported by our method is explained in Section 6.2. Section 6.3 introduces the ADDG model and briefly explains the associated equivalence checking scheme as described in [88]. Section 6.4 provides an extension of the equivalence checking scheme to handle recurrences. The correctness and the complexity of the prescribed method are formally treated in Section 6.5. The experimental results are given in Section 6.6.

The chapter is finally concluded in Section 6.7.

## 6.2 The class of supported input programs

The following restrictions characterize the input programs that can be handled by our equivalence checking method:

*Static control-flow:* A control flow of a program that can be exactly determined at compile time or by some input parameters read at the beginning of the program is called a static control flow (or data independent control flow). Programs with data dependent control flow have to be first converted into data independent control flow to make them amenable for our framework; data dependent while-loops are to be converted manually to for-loops with worst-case bounds. Restricting to static control flow is important because of non-availability of tools to determine the dependence mappings and their domains for data dependent control flow.

*Affine indices and bounds:* All loop bounds, conditions, and array index expressions of the programs must be affine expressions of the surrounding iterators and symbolic constants. They, however, need not be strictly affine, but can be piece-wise (or quasi) affine, thus, they can also have *mod, div, max, min, floor and ceil* operators. The affine indices and bounds property is critical for the decidability or computability of many of the operations that we need to perform on sets of integers and mappings described by constraints expressed as affine functions.

*Valid schedule:* To satisfy the valid-schedule restriction, each value that is read in an assignment statement has either been already written or is an input variable. This is a common requirement in most of the array data-flow analysis methods [12].

*Single-assignment form:* A behaviour is said to be in single assignment (SA) form if any memory location is written at most once in the entire behaviour. A program with static control-flow and with affine indices and bounds can automatically be converted into an equivalent program in single-assignment form [151].

*Other restrictions:* In a single-assignment form program, any variable other than the loop indices, can be replaced with an array variable [144]. Therefore, without loss of generality, it is assumed that all the statements in the behaviours involve only array variables and integer constants. Further, the program is not allowed to contain any *break, continue* or *goto statements* and *while loops*. Thus, any loop is a for-loop with three explicit parameters namely, the initial value expression, the final value expression and the modifier step of the loop iterator. The bounds are affine expressions over the surrounding loop iterators and symbolic integer parameters.

In short, the works reported in [88, 146] handle programs having the above restrictions along with the restriction on recurrences; the present work seeks to alleviate the restriction on recurrences.

## 6.3 The ADDG model and the associated equivalence checking scheme

This section introduces the ADDG model and then presents the existing equivalence checking scheme for ADDGs. The definitions and the associated explanations given here are borrowed from [88].

### 6.3.1 The ADDG model

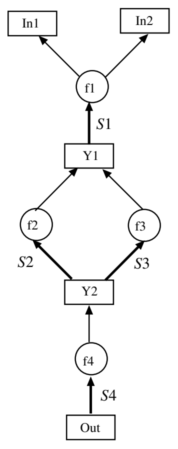
**Definition 19** (Array Data Dependence Graph (ADDG))**.** *The ADDG of a sequential behaviour is a directed graph $G = (V,E)$, where the vertex set $V$ is the union of the set $\mathcal{A}$ of array nodes and the set $\mathcal{F}$ of operator nodes and the edge set $E = \{\langle A,f\rangle \mid A \in \mathcal{A}, f \in \mathcal{F}\} \bigcup \{\langle f,A\rangle \mid f \in \mathcal{F}, A \in \mathcal{A}\}$. Edges of the form $\langle A,f\rangle$ are write edges; they capture the dependence of the left hand side (lhs) array node on the operator corresponding to the right hand side (rhs) expression. 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 $Z[\overline{e_z}] = f(Y_1[\overline{e_1}],\ldots,Y_k[\overline{e_k}])$, where $\overline{e_1},\ldots,\overline{e_k}$ and $\overline{e_z}$ are respectively the vectors of index expressions of the arrays $Y_1,\ldots,Y_k$ and $Z$, appears as a subgraph $G_S$ of $G$, where $G_S = \langle V_S,\ E_S\rangle$, $V_S = \mathcal{A}_S \bigcup \mathcal{F}_S$, $\mathcal{A}_S = \{Z,Y_1,\ldots,Y_k\} \subseteq \mathcal{A}$, $\mathcal{F}_S = \{f\} \subseteq \mathcal{F}$ and $E_S = \{\langle Z,f\rangle\} \bigcup \{\langle f,Y_i\rangle,\ 1 \le i \le k\rangle\} \subseteq E$. The write edge $\langle Z,f\rangle$ 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*

*$\langle f, Y_1 \rangle, \ldots, \langle f, Y_k \rangle$. The operator f applies over k arguments which are elements of the arrays $Y_1, \ldots, Y_k$, not necessarily all distinct.*

```
for(i = 1; i ≤ M; i = i + 1)
  for(j = 4; j ≤ N; j = j + 1)
    S1 : Y1[i + 1][j − 3] = f1(In1[i][j], In2[i][j]);

for(l = 3; l ≤ M; l = l + 1){
  for(m = 3; m ≤ N − 1; m = m + 1){
    if(l + m ≤ 7)
      S2 : Y2[l][m] = f2(Y1[l − 1][m − 2]);
    else
      S3 : Y2[l][m] = f3(Y1[l][N − 3]);
    S4 : Out[l][m] = f4(Y2[l][m]);
}}
```

(a)



(b)

Figure 6.2: (a) A nested-loop behaviour. (b) Its corresponding ADDG.

Figure 6.2 shows a sequential behaviour and its corresponding ADDG. Certain information will be extracted from each statement $S$ of the behaviour and associated with the write edge labeled with $S$ in the ADDG. Let us first consider for this purpose the generalized $x$-nested loop structure given in Figure 6.3 in which $S$ occurs. Each index (or iterator) $i_k, 1 \leq k \leq x$, has a lower limit $L_k$ and a higher limit $H_k$ and an integer (increment/decrement) step constant $r_k$. The terms $L_k$, $H_k$ are expressions of the surrounding loop iterators and integer variables. The statement $S$ executes under the condition $C_D$ over the loop indices $i_k$, $1 \leq k \leq x$, and integer constants within the loop body. All the index expressions $e_1, \ldots, e_k$ of the array $Z$ and the expressions $e'_{1,1}, \ldots, e'_{1,l_1}, \ldots, e'_{m,1}, \ldots, e'_{m,l_m}$ of the corresponding arrays $Y_1, \ldots, Y_m$ in the statement $S$ are affine arithmetic expressions over the loop indices. The related definitions follow.

```
$for(i_1 = L_1; i_1 \leq H_1; i_1 += r_1)$
  $for(i_2 = L_2; i_2 \leq H_2; i_2 += r_2)$
    ⋮
    $for(i_x = L_x; i_x \leq H_x;\ i_x += r_x)$
      $if(C_D)\ then$
          S: $Z[e_1] \ldots [e_k] = f(Y_1[e'_{1,1}] \ldots [e'_{1,l_1}], \ldots, Y_m[e'_{m,1}] \ldots [e'_{m,l_m}]);$
```

Figure 6.3: A generalized nested loop structure

**Definition 20** (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*

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

*where, for $1 \le k \le x$, the loop iterators $i_k$ are integers, $L_k, H_k$ are affine expressions over the loop iterators and some integer variables, and $r_k$ are integer constants.*

**Example 9.** Let us consider the statement $S1$ of Figure 6.2(a). 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 $I_{S2} = \{[l,\ m] \mid 3 \le l \le M \wedge 3 \le m \le N - 1 \wedge l + m \le 7 \wedge (\exists \alpha_1, \alpha_2 \in \mathbb{N} \mid l = \alpha_1 + 3 \wedge m = \alpha_2 + 3)\}$. ■

**Definition 21** (Definition mapping (${}_SM_Z^{(d)}$)). *The definition mapping of a statement $S$ describes the association between the elements of the iteration domain of the statement $S$ and the elements of its lhs array $Z$ (depicted as a suffix). ${}_SM_Z^{(d)} = I_S \rightarrow \mathbb{Z}^k$ s.t. $\forall \bar{v} \in I_S,\ \bar{v} \mapsto [e_1(\bar{v}), \ldots, e_k(\bar{v})] \in \mathbb{Z}^k$.*

The image of the function ${}_SM_Z^{(d)}$ is called the *definition domain* of the lhs array $Z$, defined as ${}_SD_Z$. So, ${}_SD_Z = {}_SM_Z^{(d)}(I_S)$. 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 ${}_SM_Z^{(d)}$ is injective (one-one). Hence, $({}_SM_Z^{(d)})^{-1}$ exists.

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

**Definition 22** (Operand mapping (${}_SM_{Y_n}^{(u)}$)). *The $n^{th}$ operand mapping of a statement $S$ describes the association between the elements of the iteration domain of the statement $S$ and the elements of one of its rhs arrays $Y_n$; specifically, ${}_SM_{Y_n}^{(u)} = I_S \rightarrow \mathbb{Z}^{l_n}$ s.t. $\forall \bar{v} \in I_S,\ \bar{v} \mapsto [e_{n,1}(\bar{v}), \ldots, e_{n,l_n}(\bar{v})] \in \mathbb{Z}^{l_n}$.*

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

**Definition 23** (Dependence mapping ($_SM_{Z,Y_n}$)). *It describes the association of the index expression of the lhs array Z which is defined through S to the index expression of the operand array $Y_n, 1 \leq n \leq m$, i.e., one of the rhs arrays of S.*

$$\begin{aligned}{}_SM_{Z,Y_n} = \ & \{[e_1,\ldots,e_k] \rightarrow [e'_1,\ldots,e'_{l_n}] \mid ([e_1,\ldots,e_k] \in {}_SD_Z \wedge \\ & [e'_1,\ldots, e'_{l_n}] \in {}_SU_{Y_n} \wedge \exists \overline{v} \in I_S \mid ([e_1,\ldots,e_k] = {}_SM^{(d)}_Z(\overline{v}) \wedge \\ & [e'_1,\ldots,e'_{l_n}] = {}_SM^{(u)}_{Y_n}(\overline{v})))\}\end{aligned}$$

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

The dependence mapping ${}_SM_{Z,Y_n}$ can be obtained by the right composition ($\diamond$)[1] of the inverse of definition mapping of $Z$ (i.e., $({}_SM^{(d)}_Z)^{-1}$) and the operand mapping of $Y_n$ (i.e., ${}_SM^{(u)}_{Y_n}$), i.e., by

$${}_SM_{Z,Y_n} = ({}_SM^{(d)}_Z)^{-1} \diamond {}_SM^{(u)}_{Y_n}$$

Specifically, ${}_SM_{Z,Y_n}({}_SD_Z) = {}_SM^{(u)}_{Y_n}(({}_SM^{(d)}_Z)^{-1}({}_SD_Z)) = {}_SM^{(u)}_{Y_n}(I_S) = {}_SU_{Y_n}$.

The *domain* of the dependence mapping ${}_SM_{Z,Y_n}$ is the definition domain of $Z$, i.e., ${}_SD_Z$. The *range* of ${}_SM_{Z,Y_n}$ is the operand domain ${}_SU_{Y_n}$. In this case, each element of ${}_SD_Z$ exactly depends on a single element of ${}_SM^{(u)}_{Y_n}$. There would be one such mapping from the defined array to each of the operand arrays in the rhs of $S$. The mappings ${}_SM_{Z,Y_n}$, $1 \leq n \leq m$, will be associated with the corresponding read edge $\langle f, Y_n \rangle$ in the ADDG.

**Example 10.** Let us consider the statement $S1$ of the Figure 6.2(a) again. For this statement,

$${}_{S1}M^{(d)}_{Y1} = \{[i,j] \rightarrow [i+1, j-3] \mid 1 \leq i \leq M \wedge 4 \leq j \leq N\}$$

$${}_{S1}D_{Y1} = \{[i+1, j-3] \mid 1 \leq i \leq M \wedge 4 \leq j \leq N\}$$

$${}_{S1}M^{(u)}_{In1} = \{[i,j] \rightarrow [i,j] \mid 1 \leq i \leq M \wedge 4 \leq j \leq N\}$$

$${}_{S1}U_{In1} = \{[i,j] \mid 1 \leq i \leq M \wedge 4 \leq j \leq N\}$$

[1] $(f \diamond g)(x) = g(f(x))$

$$
\begin{aligned}
{}_{S1}M_{Y1,In1} &= ({}_{S1}M^{(d)}_{Y1})^{-1} \diamond {}_{S1}M^{(u)}_{In1} \\
&= \{[i,j] \to [i-1,j+3] \diamond [i-1,j+3] \to [i-1,j+3]\} \mid [i,j] \in {}_{S1}D_{Y1}\} \\
&= \{[i,j] \to [i-1,j+3] \mid [i,j] \in {}_{S1}D_{Y1}\}. \qquad \blacksquare
\end{aligned}
$$

Henceforth, for brevity, we use the symbols ${}_{S}M_{\overline{Y}^{(U)}}$ and ${}_{S}M_{Z,\overline{Y}}$ to respectively represent the ordered collections (tuples) of the mappings $\langle {}_{S}M_{\overline{Y_1}^{(U)}}, \ldots, {}_{S}M_{\overline{Y_N}^{(U)}} \rangle$ and $\langle {}_{S}M_{Z,\overline{Y_1}}, \ldots, {}_{S}M_{Z,\overline{Y_N}} \rangle$; similarly, the symbol ${}_{S}U_{\overline{Y}}$ is used to represent the ordered tuple $\langle {}_{S}U_{\overline{Y_1}}, \ldots, {}_{S}U_{\overline{Y_N}} \rangle$ of the used domains of $Y_1, \ldots, Y_N$.

A data dependence exists between two statements $P$ and $Q$ if $Q$ defines the values of one array, $Y$ say, in terms of the elements of some array $Z$ and $P$ subsequently reads the same values from the array $Y$ to define another array $X$. The dependence mapping between the array $X$ and the array $Z$, i.e., ${}_{PQ}M_{X,Z}$, can be obtained from the mappings ${}_{P}M_{X,Y}$ and ${}_{Q}M_{Y,Z}$ by right composition ($\diamond$) of ${}_{P}M_{X,Y}$ and ${}_{Q}M_{Y,Z}$. The following definition captures this computation.

**Definition 24** (Transitive Dependence). ${}_{PQ}M_{X,Z} = {}_{P}M_{X,Y} \diamond {}_{Q}M_{Y,Z} = \{[i_1, \ldots, i_{l_1}] \to [k_1, \ldots, k_{l_3}] \mid \exists [j_1, \ldots, j_{l_2}]$ *s.t.* $[i_1, \ldots, i_{l_1}] \to [j_1, \ldots, j_{l_2}] \in {}_{P}M_{X,Y} \wedge [j_1, \ldots, j_{l_2}] \to [k_1, \ldots, k_{l_3}] \in {}_{Q}M_{Y,Z}\}$, *where $Y$ is used in $P$ and is defined in $Q$.*

We say that the array $Y$ satisfies "used-defined" relationship over the sequence $\langle P,\ Q \rangle$ of statements. The domain ${}_{PQ}D_X$, say, of the resultant dependence mapping (i.e., ${}_{PQ}M_{X,Z}$) is the subset of the domain of ${}_{P}M_{X,Y}$ such that ${}_{PQ}D_X = \mathit{range}({}_{P}M_{X,Y}) \bigcap \mathit{domain}({}_{Q}M_{Y,Z})$. Thus, ${}_{PQ}D_X = {}_{P}M^{-1}_{X,Y}({}_{Q}M^{-1}_{Y,Z}({}_{Q}U_Z) \bigcap {}_{P}M_{X,Y}({}_{P}D_X))$. Similarly, the range of ${}_{PQ}M_{X,Z}$ is ${}_{PQ}U_Z = {}_{Q}M_{Y,Z}({}_{P}M_{X,Y}({}_{P}D_X) \bigcap {}_{Q}M^{-1}_{Y,Z}({}_{Q}U_Z))$. The operation $\diamond$ returns empty if ${}_{P}U_Y \bigcap {}_{Q}D_Y$ is empty which indicates that the iteration domain 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 11.** Let us consider the behaviour and its corresponding ADDG of Figure 6.2. Let us now consider the statements $S4$ and $S2$ of the behaviour. We have

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

${}_{S4}D_{Out} = I_{S4}$, ${}_{S4}U_{Y2} = I_{S4}$

${}_{S4}M_{Out,Y2} = \{[l,m] \to [l,m] \mid [l,m] \in {}_{S4}D_{Out}\}$,

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

${}_{S2}D_{Y2} = I_{S2}$,

${}_{S2}U_{Y1} = \{[l-1,\ m-2] \mid [l,m] \in I_{S2}\}$ and

${}_{S2}M_{Y2,Y1} = \{[l,m] \to [l-1,m-2] \mid [l,m] \in {}_{S2}D_{Y2}\}$.

The transitive dependence mapping ${}_{S4S2}M_{Out,Y1}$ can be obtained from ${}_{S4}M_{Out,Y2}$ and ${}_{S2}M_{Y2,Y1}$ by the composition operator $\diamond$ as follows:

$$\begin{aligned}
{}_{S4S2}M_{Out,Y1} &= {}_{S4}M_{Out,Y2} \diamond {}_{S2}M_{Y2,Y1} \\
&= \{[l,m] \to [l,m] \mid [l,m] \in {}_{S4}D_{Out}\} \diamond \{[l,m] \to [l-1,m-2] \mid [l,m] \in {}_{S2}D_{Y2}\} \\
&= [l,m] \to [l-1,m-2] \mid [l,m] \in {}_{S4}D_{Out}\}. \qquad \blacksquare
\end{aligned}$$

### 6.3.2 Equivalence checking of ADDGs

Unlike a path in a control flow graph, an ADDG path may not be adequate to capture a computation corresponding to a code segment comprising even a linear sequence of assignment statements; hence we introduce the notion of a slice as follows.

**Definition 25** (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 slice g with A as the start node and* $V_1, \ldots, V_n$ *as the terminal nodes is represented as* $g(A, V_1, \ldots, V_n)$*, shortened further as* $g(A, \overline{V})$*, where* $\overline{V} = \langle V_1, \ldots, V_n \rangle$*, ordered using a consistent ordering of all the array names in the program.*

Each statement in the ADDG in Figure 6.2 represents a slice. Also, each of the statement sequences $\langle S2S1\rangle$, $\langle S3S1\rangle$, $\langle S4S2\rangle$, $\langle S4S3\rangle$, $\langle S4S2S1\rangle$ and $\langle S4S3S1\rangle$ represents a slice in this ADDG. The start array node of a slice depends on each of the terminal array nodes through a sequence of statements. Therefore, the dependence mapping between the start array node and each of the terminal array nodes of a slice can be computed as the transitive dependence mapping over the sequence of statements from the start node to the terminal node, in question, using Definition 24. The dependence mappings capture the index mappings between the start array node and the terminal array nodes in a slice. In addition, it is required to store how the output array is dependent functionally on the input arrays in a slice. We denote this notion as the *data transformation* of a slice.

**Definition 26** (Data transformation of a slice $g$ ($r_g$))**.** *It is an algebraic expression e over the terminal arrays of the slice such that e represents the value of the output array of the slice after its execution.*

The data transformation $r_g$ in a slice $g$ can be obtained by using a backward substitution method [118] 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 steps of the backward substitution method, for example, for computation of data transformation for the slice represented by statement sequence $\langle S4S2S1\rangle$ in the ADDG in Figure 6.2(b) are as follows:

$$\begin{aligned} Out &\Leftarrow f4(Y2) \quad \text{[at the node } Y2\text{]},\\ &\Leftarrow f4(f2(Y1)) \quad \text{[at the node } Y1\text{]},\\ &\Leftarrow f4(f2(f1(In1,In2))) \quad \text{[at the nodes } In1 \text{ and } In2\text{]} \end{aligned}$$

The slice $g$ is characterized by its data transformation and the list of dependence mappings between the source array and the terminal arrays.

**Definition 27** (Characteristic formula of a slice)**.** *The* characteristic formula of a slice *$g(A,\overline{V})$ is given as the tuple $\partial_g = \langle r_g, {}_gM_{A,\overline{V}}\rangle$, where A is the start node of the slice, $V_i$, $1 \le i \le n$, are the terminal nodes, $r_g$ is an arithmetic expression over $\overline{V} = \langle V_1,\ldots,V_n\rangle$ representing the data transformation of g and ${}_gM_{A,\overline{V}}$ denotes the dependence mapping between A and $\overline{V}$.*

We use the symbols ${}_gD_A$ and ${}_gU_{\overline{V}}$ to denote the definition and the operand domains of a slice.

**Definition 28** (IO-slice)**.** *A 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.*

It is required to capture the dependence of each output array on the input arrays. Therefore, IO-slices are of interest to us. It may be noted that the slices represented by the statement sequences $\langle S4S2S1\rangle$ and $\langle S4S3S1\rangle$ are the only two IO-slices in the ADDG in Figure 6.2(b). To compute the characteristic formula of a slice, the normalization technique for arithmetic expressions described in [141] is used along with some simplification rules introduced in [86, 88].

Let $G_1$ be the ADDG corresponding to an input behaviour and $G_2$ be the ADDG corresponding to the transformed behaviour obtained from the input behaviour through loop and arithmetic transformations.

**Definition 29** (Matching IO-slices)**.** *Two IO-slices $g_1$ and $g_2$ of an ADDG are said to be matching, denoted as $g_1 \approx g_2$, if the data transformations of both the slices are equivalent.*

Let the characteristic formula of $g_i(A,\overline{V})$, $i = 1,2$, be $\langle r_{g_i},\ {}_{g_i}M_{A,\overline{V}}\rangle$, where $A$ is an output array and $\overline{V} = \langle V_1,\ldots,V_l\rangle$ comprises input arrays. These two slices are matching slices if $r_{g_1} \simeq r_{g_2}$. 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 $\overline{V}$ in the slices $g_1$ and $g_2$, however, are non-overlapping.

**Definition 30** (IO-slice class)**.** *An IO-slice class is a maximum set of matching IO-slices.*

Let a slice class be $C_g(A,\overline{V}) = \{g_1,\ldots,g_k\}$. Let the characteristic formula of the member slice $g_i$, $1 \le i \le k$, be $\langle r_{g_i}, {}_{g_i}M_{A,\overline{V}}\rangle$. Due to single assignment form of the behaviour, the domain of the dependence mappings ${}_{g_i}M_{A,V_j}$, for all $i$, $1 \le i \le k$, between the output array $A$ and the input array $V_j$ in $\overline{V}$, $1 \le j \le l$, in the slices of $C_g$ must be non-overlapping. The domain of the dependence mapping ${}_{C_g}M_{A,V_j}$ from $A$ to $V_j$ over the entire class $C_g$ is the union of the domains of ${}_{g_i}M_{A,V_j}$, $1 \le i \le k$. So, the characteristic formula of the slice class $C_g$ is $\langle r_{C_g}, {}_{C_g}M_{A,\overline{V}}\rangle$, where $r_{C_g}$ is the data transformation of any of the slices in $C_g$ and ${}_{C_g}M_{A,V_j}$, $1 \le j \le l$, in ${}_{C_g}M_{A,\overline{V}}$ is

$${}_{C_g}M_{A,V_j} = \bigcup_{1 \le i \le k} {}_{g_i}M_{A,V_j}.$$

Therefore, a slice class can be visualized as a single slice.

**Definition 31** (IO-slice class equivalence:). *An IO-slice class $C_1$ of an ADDG $G_1$ is said to be equivalent to an IO-slice class $C_2$ of $G_2$, denoted as $C_1 \simeq C_2$, iff*

*(i) The data transformation of $C_1$ and $C_2$ are equivalent.*

*(ii) Both $C_1$ and $C_2$ consist of the same number of dependence mappings and the corresponding dependence mappings in the two classes are identical.*

**Definition 32** (Equivalence of ADDGs:). *An ADDG $G_1$ is said to be equivalent to an ADDG $G_2$ iff for each IO-slice class $C_1$ in $G_1$, there exists an IO-slice class $C_2$ in $G_2$ such that $C_1 \simeq C_2$, and vice-versa.*

We now give an overview of the existing equivalence checking method with an example before summarizing the broad steps in the following subsection.

```
(a)
for(k = 0; k < 64; k++){
  T1[k] = B[2 × k + 1] + C[2 × k];
  T2[k] = A[k] × T1[k]; }
for(k = 5; k < 69; k++){
  T3[k] = A[k − 5] − C[2 × k − 10];
  T4[k] = T3[k] × B[2 × k − 9]; }
for(k = 0; k < 64; k++){
  T5[k] = A[k] × C[2 × k];
  Out[k] = T2[k] − T4[k + 5] + T5[k]; }
```

```
(b)
for(k = 0; k < 64; k++){
  T[k] = 2 × A[k] + B[2 × k + 1];
  Out[k] = C[2 × k] × T[k]; }
```

Figure 6.4: (a) Original behaviour. (b) Transformed behaviour.

**Example 12.** Let us consider the behaviours of Figure 6.4 and their corresponding ADDGs in Figure 6.5. Both the programs actually compute $Out[k] = C[2k] \times (2 \times A[k] + B[2k+1])$, $0 \leq k \leq 63$. It may be noted from Figure 6.5 that each ADDG has only one IO-slice. Let the IO-slices of the source behaviour and the transformed behaviour be denoted as $s$ and $t$, respectively. The method of [88] first extracts the data transformation of the slice and the dependence mapping of each path of the slices $s$ and $t$. The data transformation $r_s$ of the slice $s$ is $A \times (B+C) - B \times (A-C) + A \times C$ and that of the slice $t$, namely $r_t$, is $C \times (2 \times A + B)$. The normalized representation of $r_s$ is $1 \times A \times B + (-1) \times A \times B + 1 \times A \times C + 1 \times A \times C + 1 \times B \times C + 0$. In this normalized expression, the terms 1 and 2 can be eliminated as the dependence

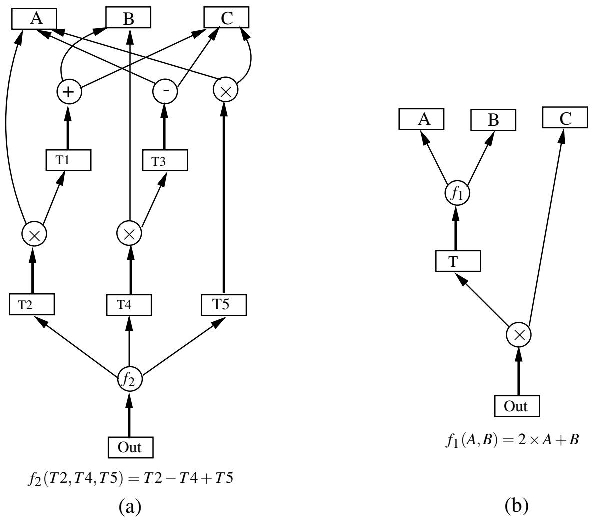

A
B
C
+
-
×
T1
T3
×
×
T2
T4
T5
$f_2$
Out
$f_2(T2, T4, T5) = T2 - T4 + T5$
(a)
A
B
C
$f_1$
T
×
Out
$f_1(A, B) = 2 \times A + B$
(b)

Figure 6.5: (a) ADDG of the original behaviour. (b) ADDG of the transformed behaviour.

mappings from the output array *Out* to the arrays $A$ and $B$ are identical. In 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 $M_{Out,B} = \{[k] \rightarrow [2 \times k + 1] \mid 0 \leq k < 64\}$ in both term 1 and term 2. Similarly, term 3 and term 4 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 $M_{Out,C} = \{[k] \rightarrow [2 \times k] \mid 0 \leq k < 64\}$, respectively. So, term 3 and term 4 can be collected. Therefore, after application of our simplification rules, $r_s$ becomes $2 \times A \times C \ + \ 1 \times B \times C \ + \ 0$. The normalized representation of $r_t$ is $2 \times A \times C \ + \ 1 \times B \times C \ + \ 0$. Therefore, $r_s \simeq r_t$. After simplification, the data transformations of the IO-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 $A$, $B$, $C^{(1)}$ in term 1 and $C^{(2)}$ in term 2. The respective dependence mappings are $M_{Out,A} = \{[k] \rightarrow [k] \mid 0 \leq k < 64\}$, $M_{Out,B} = \{[k] \rightarrow [2 \times k + 1] \mid 0 \leq k < 64\}$, $M_{Out,C^{(1)}} = \{[k] \rightarrow [2 \times k] \mid 0 \leq k < 64\}$ and $M_{Out,C^{(2)}} = \{[k] \rightarrow [2 \times k] \mid 0 \leq k < 64\}$, 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. ■

**Algorithm 9** ADDG_EQX11 (ADDG $G_1$, ADDG $G_2$)

**Inputs:** Two ADDGs $G_1$ and $G_2$.

**Outputs:** Boolean value *true* if $G_1$ and $G_2$ are equivalent, *false* otherwise; in case of failure, it reports the possible source of non-equivalence.

```
1: Find the set of IO-slices in each ADDG; find the characteristic formulae of the slices.
2: Use arithmetic simplification rule to the data transformation of the slices of $G_1$ and $G_2$.
3: Obtain the slice classes ensuring non-intersection of the dependence mapping domains of
   the constituent slices and their characteristic formula in each ADDG; let $\mathcal{H}_{G_1}$ and $\mathcal{H}_{G_2}$ be
   the respective sets of slice classes in both the ADDGs.
4: for each slice class $g_1$ in $\mathcal{H}_{G_1}$ do
5:     $g_k = findEquivalentSliceClass(g_1, \mathcal{H}_{G_2})$.
6:     /* This function returns the equivalent slice class of $g_1$ in $\mathcal{H}_{G_2}$ if found; otherwise
       returns NULL. */
7:     if $g_k = NULL$ then
8:         return false and report $g_1$ as a possible source of non-equivalence.
9:     end if
10: end for
11: Repeat the above loop by interchanging $G_1$ and $G_2$.
12: return true.
```

### 6.3.3 An overview of the method

We now explain the basic steps of the method as given in Algorithm 9.

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, commutativity and distributivity will be taken care of by the normalization process itself.

2. We then apply the simplification rule on the data transformations of the slices. The effect of other arithmetic transformations such as, 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 step 5 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 compares the respective dependence mappings.

## 6.4 Extension of the equivalence checking scheme to handle recurrences

The discussion given in the previous section clearly reveals that an ADDG without recurrence is basically a DAG which captures the data-dependence and the functional computation of an array-intensive program. Presence of recurrences introduces cycles into the ADDG and consequently, the equivalence checking strategy outlined so far fails. Consider, for example, the following code snippet:

```
$A[0] = B[0];$
$for(i = 1; i < N; i++)$
    $A[i] = A[i-1] + B[i];$
```

The array element $A[i], i \geq 0$, has the value $\sum_{j=0}^{i} B[j]$, where the array $B$ must have been defined previous to this code segment (otherwise, the program would not fulfill the requirement of having a valid schedule). As is evident from this example, it is not possible to compute the characteristic formula of a slice involving recurrence(s) using the present method. It is also not possible to obtain closed form representations of recurrences in general. Even for the restricted class of programs for which ADDGs can be constructed, it may not be always viable to obtain the corresponding closed form representations mechanically from the recurrences. As a solution, we consider separation of suitable subgraphs with cycles from the acyclic subgraphs of the ADDG. For the subgraphs with cycles, the back edges are removed to make them acyclic. Once equivalence of the resulting acyclic subgraphs from the two ADDGs is established, their corresponding original cyclic subgraphs may be replaced by identical uninterpreted functions. This process results in transforming the original ADDGs to DAGs. Now, the existing equivalence checking procedure of Karfa et al. [88], as described in the previous section, is employed to check equivalence of the transformed ADDGs. This technique to establish equivalence of subgraphs with cycles in ADDGs is now illustrated through the following example borrowed from existing literature [154].

```
S1 : A[0] = In[0];
for(i = 1; i < N; i++){
    S2 : A[i] = f(In[i]) + g(A[i - 1]);
}
S3 : Out = A[N - 1];
```

(a) Original program.

```
S1 : A[0] = In[0];
for(i = 1; i < N; i++){
    if(i%2 == 0){
        S2 : B[i] = f(In[i]);
        S3 : C[i] = g(A[i - 1]);
    } else {
        S4 : B[i] = g(A[i - 1]);
        S5 : C[i] = f(In[i]);
    }
    S6 : A[i] = B[i] + C[i];
}
S7 : Out = A[N - 1];
```

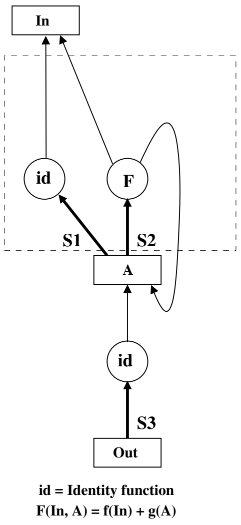
(b) Transformed program.

Figure 6.6: An example of two programs containing recurrences.

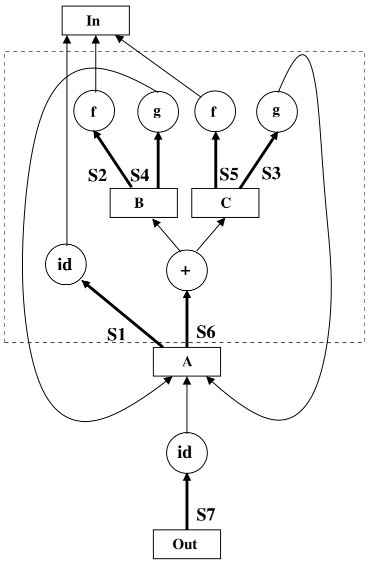
**Example 13.** Let us consider the pair of equivalent programs involving recurrences shown in Figure 6.6. The corresponding ADDGs shown in Figure 6.7 have cycles since the array $A$ has dependence upon itself.

Consider the *subgraphs containing the cycles* marked by the dotted lines in Figure 6.7(a) and Figure 6.7(b). Let the respective functional transformations computed by the subgraphs be $A[i] \Leftarrow e_1(In[i])$ and $A[i] \Leftarrow e_2(In[i]), 0 \leq i < N$. For equivalence, the equality $e_1(In[i]) = e_2(In[i])$ should hold for all $i, 0 \leq i < N$.

Suppose we proceed to prove the above equality by induction on $i$. For the basis case, therefore, $e_1(In[0]) = e_2(In[0])$ should hold. It may be noted that the slices indicated by the red edges in Figure 6.8(a) and Figure 6.8(b) depict respectively the functional transformations $A[0] \Leftarrow e_1(In[0])$ and $A[0] \Leftarrow e_2(In[0])$ and hence proving the basis case reduces to proving the equivalence of these two slice classes; we refer to such slices as *basis slices*; in fact, since, in general, there may be several cases under



(a) Original ADDG.



(b) Transformed ADDG.

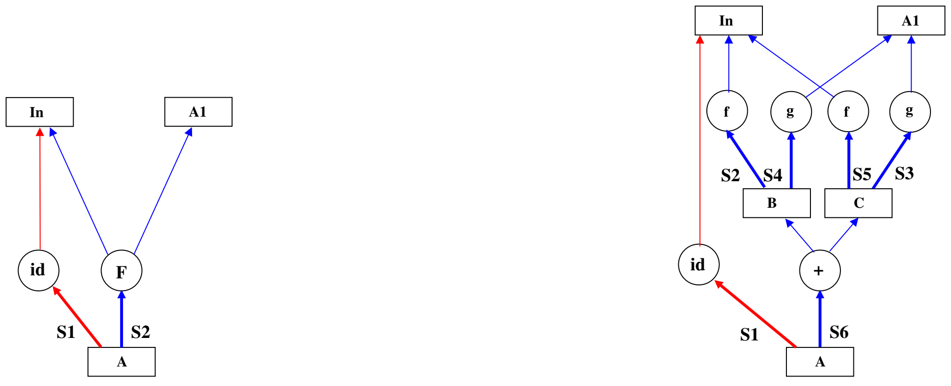
Figure 6.7: ADDGs for the programs given in Figure 6.6.

the proof of the basis step, we have *basis slice classes* together constituting a basis subgraph.

For the induction step, let the hypothesis be $e_1(In[i]) = e_2(In[i]), 0 \leq i < m$. We have to show that $e_1(In[m]) = e_2(In[m])$. Now, let the transformed array $A[0], \ldots, A[m-1]$ be designated as $A1[0], \ldots, A1[m-1]$. Specifically, the induction hypothesis permits us to assume that the parts of the array $A$ over the index range $[0, m-1]$ is identically transformed and the induction step necessitates us to show that $A1[m]$ is equivalent based on this assumption. The slices indicated by the blue edges in Figure 6.8(a) and Figure 6.8(b) capture the respective transformations of the $m$-th element (for any $m$) and proving the inductive step reduces to proving the equivalence of these two slices; accordingly, these respective slices in the two ADDGs are referred to as *induction slice classes*. The general possibility of proof of the induction step by case analysis is manifested by having several induction slice classes constituting an induction subgraph.

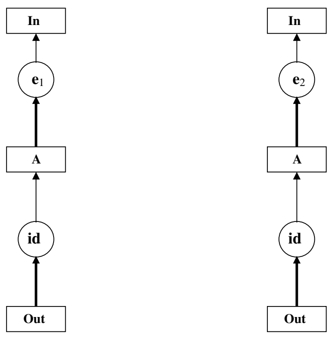
Hence the method consists of breaking the cycles by removing the backward edges identified in the depth-first traversal from the output array vertex, by incorporating a new array of the same name in both the ADDGs and then proceeding as follows.

Removing the cycles will make the resulting subADDGs given in Figure 6.8(a)



(a) Original subADDG. (b) Transformed subADDG.

Figure 6.8: Modified subADDGs corresponding to subgraphs marked by the dotted lines in Figure 6.7.



(a) Original ADDG (b) Transformed ADDG

Figure 6.9: Modified ADDGs with new uninterpreted function for the programs given in Figure 6.6.

and Figure 6.8(b) DAGs thereby permitting application of the equivalence checking scheme explained in Section 6.3.3. In the sequel, we refer to this scheme as “ADDG_EQX11” to distinguish it from its enhanced version for handling recurrences. The equivalence of these two subADDGs is established by showing the equivalence of the basis slice classes (i.e., the red edged slice classes of the subADDGs in Figure 6.8) first and then the equivalence of the induction slice classes (i.e., the blue edged slices). The first task is simple because they have identical data transformation and index domains in both the ADDGs.

The equivalence of the induction slice classes is shown as follows. For the ADDG shown in Figure 6.8(a), $M_{A,In} = \{[m] \rightarrow [m] \mid 1 \leq m < N\}$, $M_{A,A1} = \{[m] \rightarrow [m-1] \mid 1 \leq m < N\}$ and $r_{A,\{In,A1\}} = f(In) + g(A1)$. For the ADDG shown in Figure 6.8(b), $M_{B,In} = \{[m] \rightarrow [m] \mid \exists k \in \mathbb{Z}(m = 2 \times k), 1 \leq m < N\}$, $M_{C,A1} = \{[m] \rightarrow$

```
$for(i = 0; i <= 50; i++)$
  $S1 : Z[2*i] = In[i];$
$for(i = 1; i <= 99; i++)$
  $S2 : Z[i] = Z[i-1] + Z[i+1];$
```

Figure 6.10: An example where back edges exist in the absence of recurrence.

$[m-1] \mid \exists k \in \mathbb{Z}(m = 2 \times k), 1 \leq m < N\}$, $M_{B,A1} = \{[m] \to [m-1] \mid \exists k \in \mathbb{Z}(m = 2 \times k+1), 1 \leq m < N\}$, $M_{C,In} = \{[m] \to [m] \mid \exists k \in \mathbb{Z}(m = 2 \times k+1), 1 \leq m < N\}$, $M_{A,B} = \{[m] \to [m] \mid 1 \leq m < N\}$, $M_{A,C} = \{[m] \to [m] \mid 1 \leq m < N\}$. Now we find that $r^{(1)}_{A,\{In,A1\}} = f(In) + g(A1)$ for the domain $\{[m] \mid \exists k \in \mathbb{Z}(m = 2 \times k), 1 \leq m < N\}$ and $r^{(2)}_{A,\{In,A1\}} = f(In) + g(A1)$ for the domain $\{[m] \mid \exists k \in \mathbb{Z}(m = 2 \times k+1), 1 \leq m < N\}$; (the superfixes refer to the data transformations of the array elements referred through the terms in the rhs expressions;) note that our normalization technique accommodates the commutativity of the '$+$' operation. Since the data transformation is identical in both the domains, they constitute a slice class with domain $\{[m] \mid 1 \leq m < N\}$. Hence, we arrive at the same mapping and data transformation as that of Figure 6.8(a). Note that the method automatically extracts two pairs of slice classes from the subADDGs in Figure 6.8 – one corresponding to the basis cases and the other corresponding to the inductions.

Having established that the two subgraphs with cycles in the respective ADDGs are equivalent, we construct another pair of modified ADDGs as shown in Figure 6.9. The ADDGs in Figure 6.9(a) and Figure 6.9(b) contain the new uninterpreted functions $e_1$ and $e_2$, respectively, where $e_1 = e_2$. It is to be noted that scalar variables, such as *Out*, are treated as array variables of unit dimension, i.e., *Out* is considered as *Out*[0]. Now showing equivalence of the two entire ADDGs of Figure 6.9 (which are basically DAGs) is straightforward by the ADDG equivalence checking method (ADDG_EQX11) given in Section 6.3.3. ■

Before formalizing the above mechanism, we underline the fact that while recurrences imply back edges in an ADDG, the converse is not true; we may have back edges even when there is no recurrence. This is illustrated through the following example.

**Example 14.** *Consider the program segment given in Figure 6.10. While constructing the ADDG corresponding to this program, there will be two back edges in the ADDG for statement $S2$ corresponding to the two rhs terms $Z[i-1]$ and $Z[i+1]$; however, note* *that statement $S2$ does not have any data dependency on itself since the whole of its operand domain has already been defined in statement $S1$. These cases can be handled within the framework of the equivalence checking method described in Section 6.3.3 by permitting computation of transitive dependence through the cycles only once and then continuing through the other edges emanating from the array vertex $Z$.*

Example 14 leads us to the following definition.

**Definition 33** (Recurrence Array Vertex)**.** *An array vertex $Z$ which is identified as the destination of a back edge (of a cycle $c$) during a depth-first traversal of an ADDG starting from an output array such that ${}_cD_Z \bigcap {}_cU_Z \neq \emptyset$ is called a* recurrence array vertex.

Let us first elaborate what the above definition entails. Obviously, recurrences lead to cycles and a back edge basically identifies the source of the recurrence namely, the array $Z$ which has been defined in terms of itself. Let the statement leading to the recurrence in some program be
$S : Z[l] = f(Y[r_0], Z[r_1], Z[r_2], Z[r_3]);$
Let the symbol $Z^{(i)}$ represent the $i$-th occurrence of $Z$ in the rhs of $S$. For $Z$ to qualify as a recurrence array vertex, $({}_SM^{(d)}_{Z,Z^{(1)}}(I_S) = {}_SD_Z) \bigcap ({}_SM^{(u)}_{Z,Z^{(1)}}(I_S) = {}_SU_{Z^{(1)}}) \neq \emptyset$ or ${}_SD_Z \bigcap {}_SU_{Z^{(2)}} \neq \emptyset$ or ${}_SD_Z \bigcap {}_SU_{Z^{(3)}} \neq \emptyset$ should hold (other arrays such as, $Y$, do not participate in this procedure). In case the recurrence occurs through a cycle $c$ involving multiple statements, we shall have to consider the transitive dependences over $c$. This check for overlapping definition and operand domains helps in segregating cases of *true* recurrences from those as shown in Example 14.

As indicated in Example 13, for each recurrence array vertex, we need to find a minimum subgraph with cycle(s) so that the computed values of the recurrence array elements can be captured by an uninterpreted function with proper arguments. Finding such minimum subgraphs with cycles in an ADDG involves prior identification of strongly connected components (SCCs) in the ADDG.

**Definition 34** (Basis Subgraph $\mathcal{B}(Z, \{Y^i_1, \dots, Y^i_{k_i}, k_i \geq 1, 1 \leq i \leq m\})$ for the recurrence array vertex $Z$)**.** *Let $\mathcal{C}(Z)$ be an SCC in the ADDG having $Z$ as the recurrence array vertex. Let $e_i = \langle Z, f_i \rangle, 1 \leq i \leq m$, be $m$ write edges which are not contained in $\mathcal{C}(Z)$; also, let $\langle f_i, Y^i_1 \rangle, \dots, \langle f_i, Y^i_{k_i} \rangle, k_i \geq 1$, be all the existing read edges emanating from the operator vertex $f_i$ in the ADDG. The subgraph $\mathcal{B}(Z, \{Y^i_1, \dots, Y^i_{k_i}, k_i \geq$*

$1, 1 \leq i \leq m\}$) *containing the array vertex* $Z$*, all such operator vertices* $f_i, 1 \leq i \leq m$*, and the corresponding array vertices* $Y_1^i, \ldots, Y_{k_i}^i$ *along with the connecting edges* $\langle Z, f_i \rangle, \langle f_i, Y_1^i \rangle, \ldots, \langle f_i, Y_{k_i}^i \rangle$ *is called a* basis subgraph*. For brevity, we represent such a basis subgraph as* $\mathcal{B}(Z, \overline{Y})$*, where* $\overline{Y}$ *represents the set* $\{Y_1^i, \ldots, Y_{k_i}^i, k_i \geq 1, 1 \leq i \leq m\}$ *ordered using a consistent ordering of all the array names used in the program.*

Note that the subgraph $\mathcal{B}(Z, \overline{Y})$ is the minimum subgraph that encompasses the basis step(s) of a recurrence involving the recurrence array $Z$ in the program. The arrays $Y_j^i, 1 \leq j \leq k_i, 1 \leq i \leq m$, in $\overline{Y}$ need not be all distinct. If there are multiple basis steps with different data transformations, then there will be multiple basis slice classes; it is to be noted that all such basis slice classes are collectively covered in $\mathcal{B}(Z, \overline{Y})$.

**Definition 35** (Induction Subgraph $\mathcal{D}(Z, \{T_1^i, \ldots, T_{k_i}^i, k_i \geq 1, 1 \leq i \leq m\}, \{W_1^i, \ldots, W_{k_i}^i, k_i \geq 1, 1 \leq i \leq n\})$ for the recurrence array vertex $Z$)**.** *Let* $C(Z)$ *be an SCC in the ADDG having* $Z$ *as the recurrence vertex.*
*1) Let* $e_i = \langle X_i, h_i \rangle, 1 \leq i \leq m$*, be* $m$ *write edges where* $X_i (\neq Z)$ *is an array vertex in* $C(Z)$ *such that* $e_i$ *is not contained in* $C(Z)$*; also, let* $\langle h_i, T_1^i \rangle, \ldots, \langle h_i, T_{k_i}^i \rangle, k_i \geq 1, 1 \leq i \leq m$*, be all the existing read edges emanating from the operator vertex* $h_i$ *in the ADDG. Let the connected subgraph containing* $C(Z)$ *along with the vertices* $h_i, T_1^i, \ldots, T_{k_i}^i, k_i \geq 1, 1 \leq i \leq m$*, be designated as* $\mathcal{D}'(Z, T_1^i, \ldots, T_{k_i}^i, k_i \geq 1, 1 \leq i \leq m)$*.*
*2) Let* $g_i, 1 \leq i \leq n$*, be* $n$ *operator vertices in* $C(Z)$ *such that there is a read edge* $e$ *emanating from* $g_i$ *which is not contained in* $C(Z)$*; also, let* $\langle g_i, W_1^i \rangle, \ldots, \langle g_i, W_{k_i'}^i \rangle, k_i' \geq 1, 1 \leq i \leq n$*, be all the existing read edges emanating from the operator vertex* $g_i$ *in the ADDG which are not already covered in* $C(Z)$*; the connected subgraph containing* $\mathcal{D}'(Z, T_1^i, \ldots, T_{k_i}^i, k_i \geq 1, 1 \leq i \leq m)$ *along with all such array vertices* $W_1^i, \ldots, W_{k_i'}^i, k_i' \geq 1, 1 \leq i \leq n$*, is called an* induction subgraph *and is represented as* $\mathcal{D}(Z, \overline{V})$*, where* $\overline{V}$ *is the set* $\{T_1^i, \ldots, T_{k_i}^i, k_i \geq 1, 1 \leq i \leq m\} \bigcup \{W_1^i, \ldots, W_{k_i'}^i, k_i' \geq 1, 1 \leq i \leq n\}$) *ordered using a uniform ordering of all the array names used in the program.*

Note that the subgraph $\mathcal{D}(Z, \overline{V})$ is the minimum subgraph that encompasses the induction step(s) of a recurrence involving the array $Z$ in the program. Once the induction subgraph is found, we do not need to distinguish between the arrays marked as $T$'s and $W$'s; hence they are jointly denoted as $\overline{V}$. The arrays $T_j^i, 1 \leq j \leq k_i, 1 \leq i \leq m$, and $W_j^i, 1 \leq j \leq k_i, 1 \leq i \leq n$, in $\overline{V}$ need not be all distinct. If there are multiple induction steps with different data transformations, then there will be multiple induction slice classes; it is to be noted that all such induction slice classes are collectively covered in $\mathcal{D}(Z,\overline{V})$.

**Definition 36** (Recurrence Subgraph $\mathcal{E}(Z,\overline{Y},\overline{V}\})$ for the basis subgraph $\mathcal{B}(Z,\overline{Y})$ and the induction subgraph $\mathcal{D}(Z,\overline{V})$)**.** *The subgraph $\mathcal{E}(Z,\overline{Y},\overline{V})$ which is obtained by combining a basis subgraph $\mathcal{B}(Z,\overline{Y})$ and an induction subgraph $\mathcal{D}(Z,\overline{V})$ having the same recurrence array vertex Z in an ADDG is called a* recurrence subgraph.

Note that the subgraph $\mathcal{E}(Z,\overline{Y},\overline{V})$ is the minimum subgraph that encompasses the recurrence involving the array $Z$ in the program.

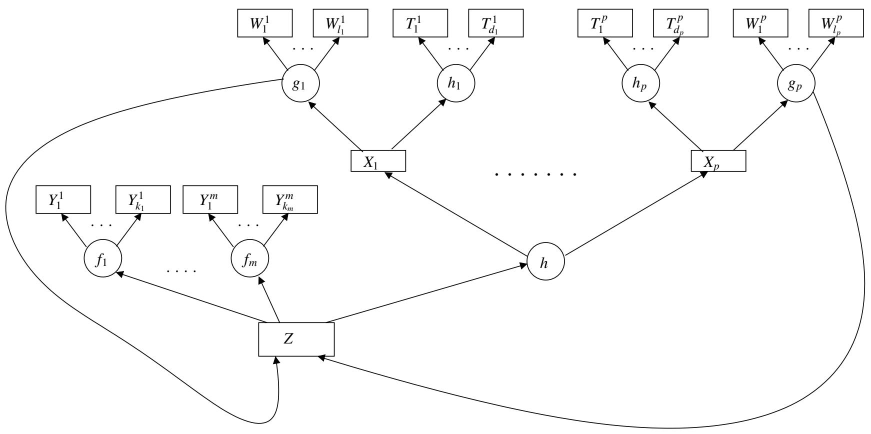
Figure 6.11(a) shows a schema of a program involving a recurrence and Figure 6.11(b) shows its ADDG representation; note that we have shown a single back edge from the operator vertices $g_1$ and $g_p$ instead of $m$ such back edges for clarity; the corresponding basis subgraph and the induction subgraph have been given in Figure 6.12(a) and Figure 6.12(b), respectively.

In order to compare two recurrence subgraphs, $\mathcal{E}_1(Z,\overline{Y_1},\overline{V_1})$ and $\mathcal{E}_2(Z,\overline{Y_2},\overline{V_2})$, say, from two different ADDGs, and represent them as uninterpreted functions subsequently, the arguments $\overline{Y}$ and $\overline{V}$ must be identical, i.e., $\overline{Y_1} = \overline{Y_2}$ and $\overline{V_1} = \overline{V_2}$ have to hold. In case they do not hold, we identify the uncommon arrays, i.e., arrays that appear in only one of the ADDGs but not both, occurring in $\overline{Y_i}$ and $\overline{V_i}, i \in \{1,2\}$; for each such array, $T$ say, we identify the minimum subgraph $G_T(T,\overline{X_T})$, say (with $T$ as the start array vertex and $\overline{X_T}$ as the terminal array vertices), where $\overline{X_T}$ comprises only *common arrays*, i.e., arrays that appear in both the ADDGs. If $T \in \overline{Y_i}$ ($\overline{V_i}$), then the basis subgraph $\mathcal{B}_i(Z,\overline{Y_i})$ (induction subgraph $\mathcal{D}_i(Z,\overline{V_i})$) is extended by including $G_T$ and treat the resulting subgraph as the basis subgraph (induction subgraph) for establishing equivalence of the two ADDGs. Algorithm 10 captures this process, where the module "extendSubgraph" basically extends the passed recurrence subgraph along each uncommon array vertex $A$ by including all edges of the form $\langle A, f_i\rangle$ and $\langle f_i, X_i\rangle, 1 \leq i \leq p,$; this process of extension is repeated for each uncommon array vertex $X_i, 1 \leq i \leq p$, until each of the terminal nodes of the extended subgraph is a common array vertex.

For example, from Figure 6.7(b), we shall get an SCC $\mathcal{C}(A)$, say, comprising the vertices ($A$, $+$, $B$, $C$, $g_{S4}$, $g_{S3}$) with $A$ designated as the recurrence array vertex; the corresponding basis subgraph $\mathcal{B}(A,In)$ comprises the vertices $\{A$, $id_{S1}$, $In\}$ and the

```
$Z[0] = f_1(Y_1^1[j_1^1], \ldots, Y_{k_1}^1[j_{k_1}^1]);$
$\vdots$
$Z[m] = f_m(Y_1^m[j_1^m], \ldots, Y_{k_m}^m[j_{k_m}^m]);$
$for(i = m+1, i \leq N; i++)$ {
  $if(\ Cond(i)\ )$ {
    $X_1[i] = g_1(Z[0], \ldots, Z[m], W_1^1[i], \ldots, W_{l_1}^1[i]);$
    $\vdots$
    $X_p[i] = g_p(Z[0], \ldots, Z[m], W_1^p[i], \ldots, W_{l_p}^p[i]);$
  } $else$ {
    $X_1[i] = h_1(T_1^1[i], \ldots, T_{d_1}^1[i]);$
    $\vdots$
    $X_p[i] = h_p(T_1^p[i], \ldots, T_{d_p}^p[i]);$
  }
  $Z[i] = h(X_1[i], \ldots, X_p[i]);$
}
```

(a)



(b)

Figure 6.11: (a) Original program. (b) Corresponding ADDG.

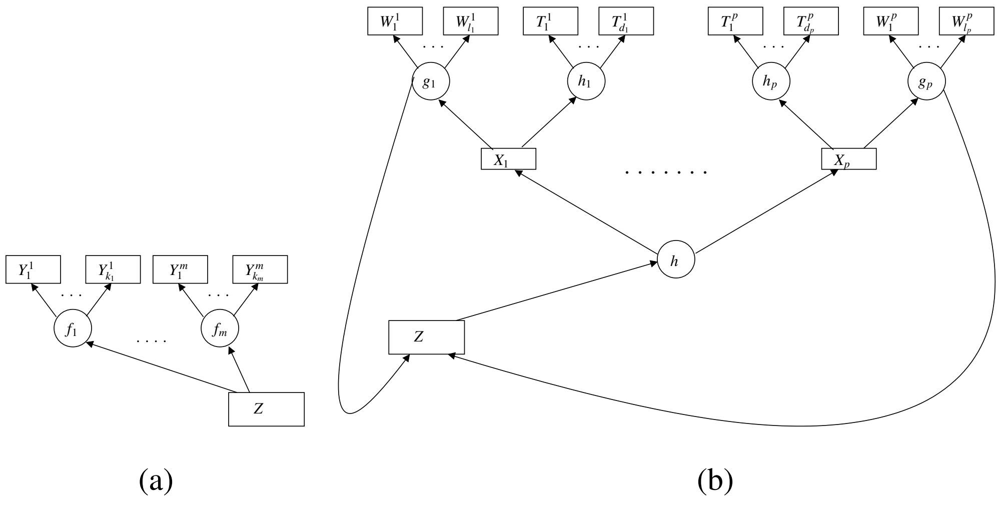


Figure 6.12: (a) Basis subgraph. (b) Induction subgraph corresponding to Figure 6.11.

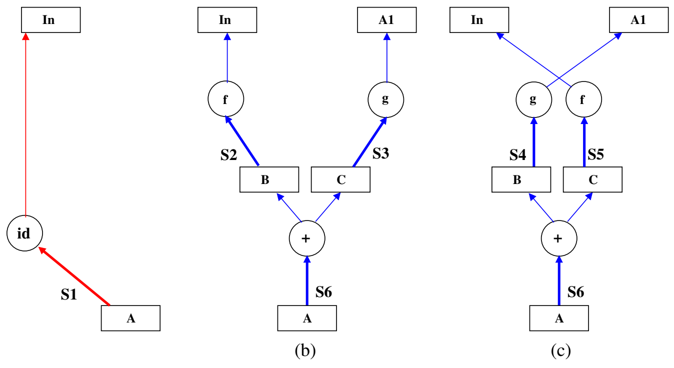


Figure 6.13: (a) Basis slice. (b) Valid induction slice 1. (c) Valid induction slice 2, corresponding to Figure 6.8(b).

**Algorithm 10** extendRecurrenceSubgraph (ADDG $G$, Subgraph $\mathcal{E}(Z,\overline{Y},\overline{V})$, List $L$)

**Inputs:** An ADDG $G$, a recurrence subgraph $\mathcal{E}(Z,\overline{Y},\overline{V})$ and a list $L$ of common arrays (including the input and the output arrays).

**Outputs:** An extended recurrence subgraph $\mathcal{E}'(Z,\overline{Y}',\overline{V}')$ whose each terminal array vertex belonging to $\overline{Y}'$ or $\overline{V}'$ is a common array belonging to $L$.

```
1: Set $\mathcal{A} \leftarrow \{\overline{Y}\} \bigcup \{\overline{V}\}$.
2: Let $\overline{Y}' \leftarrow \overline{Y}$; $\overline{V}' \leftarrow \overline{V}$; $\mathcal{B}'(Z,\overline{Y}') \leftarrow \mathcal{B}(Z,\overline{Y})$ and $\mathcal{D}'(Z,\overline{V}') \leftarrow \mathcal{D}(Z,\overline{V})$.
3: for all $T \in (\mathcal{A} - L)$ do
4:     $G_T(T,\overline{X_T}) \leftarrow$ extendSubgraph ($G$, $\mathcal{E}$, $T$).
5:     if $T \in \overline{Y}$ then
6:         Let $\overline{Y}' \leftarrow \overline{Y}'\{T \leftarrow \overline{X_T}\}$;
           /* In the ordered tuple $\overline{Y}'$, each member of $\overline{T}$ is substituted with a subtuple of $\overline{X_T}$ and
           then the whole tuple is ordered using the consistent ordering of all the array names
           of $L$. */
           $\mathcal{B}'(Z,\overline{Y}') \leftarrow$ compose ($\mathcal{B}'(Z,\overline{Y}')$, $G_T(T,\overline{X_T})$).
7:     else
8:         Let $\overline{V}' \leftarrow \overline{V}'\{U \leftarrow \overline{X_U}\}$;
           $\mathcal{D}'(Z,\overline{V}') \leftarrow$ compose ($\mathcal{D}'(Z,\overline{V}')$, $G_T(T,\overline{X_T})$).
9:     end if
10: end for
11: Let $\overline{Y} \leftarrow \overline{Y}'$; $\overline{V} \leftarrow \overline{V}'$;
    $\mathcal{E}'(Z,\overline{Y}',\overline{V}') \leftarrow$ compose ($\mathcal{B}'(Z,\overline{Y}')$, $\mathcal{D}'(Z,\overline{V}')$).
12: return $\mathcal{E}'(Z,\overline{Y}',\overline{V}')$.
```

connecting edges; the corresponding induction subgraph $\mathcal{D}(A,In)$ comprises $\mathcal{C}(A)$, the vertices $\{f_{S_2}, f_{S_5}, In\}$ and the connecting edges. The recurrence subgraph $\mathcal{E}(A,In)$ is obtained by taking a union of the vertices and edges in $\mathcal{B}(A,In)$ and $\mathcal{D}(A,In)$. Although $B$ and $C$ are uncommon arrays, extraction of the minimum induction subgraph itself will not identify them as terminal arrays due to clause (1) of Definition 35; no extra extension step will be needed.

The method initially finds five slices corresponding to the subADDG given in Figure 6.8(b) as given below (in terms of the involved statements): $g_1 = \langle S1 \rangle$, $g_2 = \langle S6,S2,S5 \rangle$, $g_3 = \langle S6,S2,S3 \rangle$, $g_4 = \langle S6,S4,S5 \rangle$, $g_5 = \langle S6,S4,S3 \rangle$. However, out of these five slices, slices $g_2$ and $g_5$ are deemed invalid because they contain conflicting conditions, namely $i\%2\ ==\ 0$ and $i\%2\ !=\ 0$, in the dependence mapping ${}_{g_2}M_{A,In}$ and ${}_{g_5}M_{A,A1}$, respectively. Thus, the validity of an induction slice is to be checked before it is added to the respective set of induction slice classes. Note that the function module “compose” prunes such invalid slices. The basis slice and the valid induction slices have been shown in Figure 6.13.

It is to be noted that the method of [154] resolves the equivalence of the data transformations of the two programs shown in Figure 6.6 by comparing all possible permutations of the involved commutative operator ($+$ in this case). This method, however, would not have been successful in establishing equivalence if the statement $S2$ in Figure 6.6(a) had been replaced by $S2 : A[i] = f(In[i]) + g(A[i-1]) + 4$; and 2 had been added to each of the rhs expressions in the statements $S2, S3, S4$ and $S5$ in Figure 6.6(b) (e.g., statement $S2$ in Figure 6.6(b) was replaced by $S2 : B[i] = f(In[i]) + 2$;). Our method, unlike that of [154] which only checks syntactic equivalence of the operands, can show the equivalence even under such a scenario since it employs the normalization technique of [88] to check equivalence of arithmetic transformations.

Our overall equivalence checking method is now presented in Algorithm 11, where the function module “getRecurrenceSubgraph” is responsible for identifying true recurrences and excluding those SCCs which may have been generated by cases such as in Example 14; the module “obtainDAG” takes a (cyclic) recurrence subgraph $\mathcal{E}(Z,\overline{Y},\overline{V})$ as input, produces a subgraph $\mathcal{E}'(Z,\overline{Y},\overline{V},Z1)$ by replacing all back edges of the form $\langle f,Z\rangle$ by $\langle f,Z1\rangle$ in $\mathcal{E}'$ and copies the dependence mappings ${}_{\mathcal{E}_1}M_{Z,Z}$ as ${}_{\mathcal{E}'_1}M_{Z,Z1}$, where $Z1$ is a new array vertex (not already present in the ADDG); the function module “replaceRecurrenceSubgraphByUF” takes an ADDG $G$, a recurrence subgraph $\mathcal{E}(Z,\overline{Y},\overline{V})$ and an uninterpreted function $f$ as input and replaces the recurrence subgraph $\mathcal{E}(Z,\overline{Y},\overline{V})$ in $G$ by a directed acyclic subgraph $\mathcal{H}(Z,\overline{Y},\overline{V})$ having a start vertex $Z$, terminal vertices $\overline{Y}$ and $\overline{V}$, a single operator $f$, write edge $\langle Z,f\rangle$ and a set $\{\langle f,Y_i\rangle, Y_i \in \overline{Y}\} \bigcup \{\langle f,V_i\rangle, V_i \in \overline{V}\}$ of read edges; the mappings ${}_{\mathcal{H}}M_{Z,\overline{Y}}$ and ${}_{\mathcal{H}}M_{Z,\overline{V}}$ are kept the same as ${}_{\mathcal{E}}M_{Z,\overline{Y}}$ and ${}_{\mathcal{E}}M_{Z,\overline{V}}$, respectively.

**Algorithm 11** equivalenceChecker (ADDG $G_1$, ADDG $G_2$)

**Inputs:** Two ADDGs $G_1$ and $G_2$.

**Outputs:** Boolean value *true* if $G_1$ and $G_2$ are equivalent, *false* otherwise; in case of failure, it reports the possible source of non-equivalence.

```
1:  Set $L \leftarrow$ findCommonArrays ($G_1$, $G_2$); $G'_i \leftarrow G_i$, $i \in \{1,2\}$.
2:  Set $\mathcal{C}_i$ of SCCs $\leftarrow$ findStronglyConnectedComponents ($G_i$), $i \in \{1,2\}$.
3:  for each SCC $c_1(Z_1) \in \mathcal{C}_1$ do
4:      $e_1(Z_1,\overline{Y_1},\overline{V_1}) \leftarrow$ getRecurrenceSubgraph ($G_1$, $c_1(Z_1)$);
        $e'_1(Z_1,\overline{Y_1}',\overline{V_1}') \leftarrow$ extendRecurrenceSubgraph ($G_1$, $e_1(Z_1,\overline{Y_1},\overline{V_1})$, $L$);
        $e''_1(Z_1,\overline{Y_1}',\overline{V_1}',Z1_1) \leftarrow$ obtainDAG ($e'_1(Z_1,\overline{Y_1}',\overline{V_1}')$).
5:  end for
6:  for each $c_2(Z_2) \in \mathcal{C}_2$ do
7:      $e_2(Z_2,\overline{Y_2},\overline{V_2}) \leftarrow$ getRecurrenceSubgraph ($G_2$, $c_2(Z_2)$);
        $e'_2(Z_2,\overline{Y_2}',\overline{V_2}') \leftarrow$ extendRecurrenceSubgraph ($G_2$, $e_2(Z_2,\overline{Y_2},\overline{V_2})$, $L$);
        $e''_2(Z_2,\overline{Y_2}',\overline{V_2}',Z1_2) \leftarrow$ obtainDAG ($e'_2(Z_2,\overline{Y_2}',\overline{V_2}')$).
8:  end for
9:  for each $e''_1(Z_1,\overline{Y_1}',\overline{V_1}',Z1_1)$ do
10:     for each $e''_2(Z_2,\overline{Y_2}',\overline{V_2}',Z1_2)$ do
11:         if $Z_1 = Z_2 \wedge \overline{Y_1}' = \overline{Y_2}' \wedge \overline{V_1}' = \overline{V_2}' \wedge$
              ADDG_EQX11 ($e''_1(Z_1,\overline{Y_1}',\overline{V_1}',Z1_1)$, $e''_2(Z_2,\overline{Y_2}',\overline{V_2}',Z1_2)$) $=$ true then
12:             $G'_1 \leftarrow$ replaceRecurrenceSubgraphByUF ($G'_1$, $e'_1(Z_1,\overline{Y_1}',\overline{V_1}')$, $f$). /* where $f$ is some new
                uninterpreted function */
13:             $G'_2 \leftarrow$ replaceRecurrenceSubgraphByUF ($G'_2$, $e'_2(Z_2,\overline{Y_2}',\overline{V_2}')$, $f$). /* here $f$ is the same
                uninterpreted function as in the previous step */
14:         end if
15:     end for
16:     if no match for $e''_1(Z_1,\overline{Y_1}',\overline{V_1}',Z1_1)$ is found then
17:         return false and report $e''_1(Z_1,\overline{Y_1}',\overline{V_1}',Z1_1)$ as a possible source of non-equivalence.
18:     end if
19: end for
20: if some $e''_2(Z_2,\overline{Y_2}',\overline{V_2}',Z1_2)$ exists which is not found to have equivalence with any
        $e''_1(Z_1,\overline{Y_1}',\overline{V_1}',Z1_1)$ then
21:     return false and report $e''_2(Z_2,\overline{Y_2}',\overline{V_2}',Z1_2)$ as a possible source of non-equivalence.
22: end if
23: if ADDG_EQX11 ($G'_1$, $G'_2$) $=$ true then
24:     return true.
25: else
26:     return false and report the slice class in an ADDG which has no equivalent slice class in the
        other ADDG.
27: end if
```

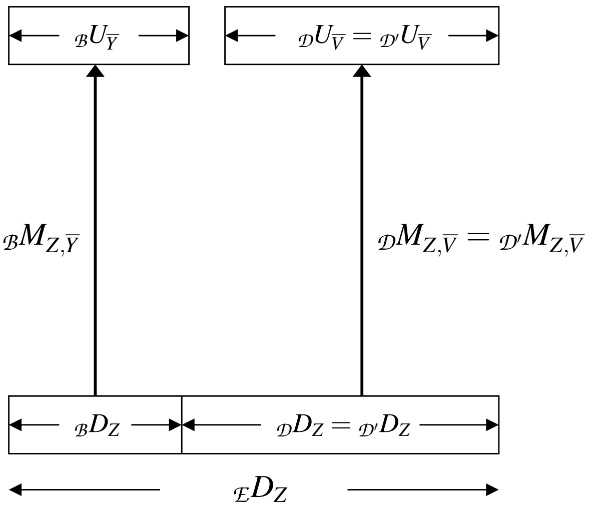

${}_{\mathcal{B}}U_{\overline{Y}}$
${}_{\mathcal{D}}U_{\overline{V}} = {}_{\mathcal{D}'}U_{\overline{V}}$
${}_{\mathcal{B}}M_{Z,\overline{Y}}$
${}_{\mathcal{D}}M_{Z,\overline{V}} = {}_{\mathcal{D}'}M_{Z,\overline{V}}$
${}_{\mathcal{B}}D_Z$
${}_{\mathcal{D}}D_Z = {}_{\mathcal{D}'}D_Z$
${}_{\mathcal{E}}D_Z$

Figure 6.14: Relationship between different domains.

## 6.5 Correctness and complexity

### 6.5.1 Correctness

**Theorem 10** (Soundness). *Let $\mathcal{E}_1(Z,\overline{Y},\overline{V})$ be a recurrence subgraph of the ADDG $G_1$ of the source program $P_1$ and $\mathcal{E}_2(Z,\overline{Y},\overline{V})$ be a recurrence subgraph of the ADDG $G_2$ of the transformed program $P_2$ with identical parameters $Z,\overline{Y}$ and $\overline{V}$. Let $\mathcal{E}_1(Z,\overline{Y},\overline{V})$ have the basis subgraph $\mathcal{B}_1(Z,\overline{Y})$ and the induction subgraph $\mathcal{D}_1(Z,\overline{V})$; similarly, let $\mathcal{B}_2(Z,\overline{Y})$ and $\mathcal{D}_2(Z,\overline{V})$ respectively be the basis and the induction subgraphs of $\mathcal{E}_2(Z,\overline{Y},\overline{V})$. Let $\Sigma_1 = \{\overline{Y}[d_{\overline{Y}}], d_{\overline{Y}} \in {}_{\mathcal{B}_1}U_{\overline{Y}}$, the used domain of $\overline{Y}$ in $\mathcal{B}_1\} \biguplus \{\overline{V}[d_{\overline{V}}], d_{\overline{V}} \in {}_{\mathcal{D}_1}U_{\overline{Y}}$, the used domain of $\overline{V}$ in $\mathcal{D}_1\}$. Similarly, let $\Sigma_2 = \{\overline{Y}[d_{\overline{Y}}], d_{\overline{Y}} \in {}_{\mathcal{B}_2}U_{\overline{Y}}$, the used domain of $\overline{Y}$ in $\mathcal{B}_2\} \biguplus \{\overline{V}[d_{\overline{V}}], d_{\overline{V}} \in {}_{\mathcal{D}_2}U_{\overline{Y}}$, the used domain of $\overline{V}$ in $\mathcal{D}_2\}$. Let the function defined over the elements of $\Sigma_1$ yielding values for $Z[d], d \in {}_{\mathcal{E}_1}D_Z$, be represented as $e_1$ and the function defined over the elements of $\Sigma_2$ yielding values for $Z[d], d \in {}_{\mathcal{E}_2}D_Z$, be represented as $e_2$. Let the directed acyclic (DA)-ADDGs corresponding to the subgraphs $\mathcal{D}_1(Z,\overline{V})$ and $\mathcal{D}_2(Z,\overline{V})$ be $\mathcal{D}'_1(Z,\overline{V},Z1)$ and $\mathcal{D}'_2(Z,\overline{V},Z1)$, respectively. If the equivalence checker ADDG_EQX11 ascertains that $\mathcal{B}_1(Z,\overline{Y}) \simeq \mathcal{B}_2(Z,\overline{Y})$ and $\mathcal{D}'_1(Z,\overline{V},Z1) \simeq \mathcal{D}'_2(Z,\overline{V},Z1)$, then $e_1 = e_2$.*

*Proof:* Figure 6.14 shows the relationship between different domains in a recurrence subgraph. Consider any element $d \in {}_{\mathcal{E}_1}D_Z$. By the single assignment (SA) property of the program $P_1$, ${}_{\mathcal{B}_1}D_Z$ and ${}_{\mathcal{D}_1}D_Z$ constitute a partition of ${}_{\mathcal{E}_1}D_Z$; hence, we have the following two *mutually exclusive* cases: (1) $d \in {}_{\mathcal{B}_1}D_Z$ and (2) $d \in {}_{\mathcal{D}_1}D_Z$. We carry out proof by cases.

*Case 1 [$d \in {}_{\mathcal{B}_1}D_Z$]:* Here,

$$\begin{aligned}
Z[d] &= e_1(\overline{Y}[d_{\overline{Y}}]),\ \text{for some } d_{\overline{Y}} = {}_{\mathcal{B}_1}M_{Z,\overline{Y}}(d) \in {}_{\mathcal{B}_1}U_{\overline{Y}} \\
&= r_{\mathcal{B}_1}(\overline{Y}[d_{\overline{Y}}]),\ \text{where } r_{\mathcal{B}_1} \text{ is the data transformation of the subgraph } \mathcal{B}_1 \\
&= r_{\mathcal{B}_2}(\overline{Y}[d'_{\overline{Y}}]),\ \text{for some } d'_{\overline{Y}} = {}_{\mathcal{B}_2}M_{Z,\overline{Y}}(d') \in {}_{\mathcal{B}_2}U_{\overline{Y}} \\
&= e_2(\overline{Y}[d'_{\overline{Y}}]),\ \text{since the equivalence checker ADDG\_EQX11 ascertains} \\
&\quad \mathcal{B}_1(Z,\overline{Y}) \simeq \mathcal{B}_2(Z,\overline{Y}),\ \text{it must have found } r_{\mathcal{B}_1} = r_{\mathcal{B}_2} \text{ over the domain} \\
&\quad \{\overline{Y}[d_{\overline{Y}}], d_{\overline{Y}} \in {}_{\mathcal{B}_1}U_{\overline{Y}} = {}_{\mathcal{B}_2}U_{\overline{Y}}\} \text{ and} \\
&\quad {}_{\mathcal{B}_1}M_{Z,\overline{Y}} = {}_{\mathcal{B}_2}M_{Z,\overline{Y}} \text{ over the domains } {}_{\mathcal{B}_1}D_Z = {}_{\mathcal{B}_1}D_Z
\end{aligned}$$

Thus, $d_{\overline{Y}} = {}_{\mathcal{B}_1}M_{Z,\overline{Y}}(d) = {}_{\mathcal{B}_2}M_{Z,\overline{Y}}(d') = d'_{\overline{Y}}$. Hence, from the soundness of DA-ADDG equivalence checker, $e_1(\overline{Y}[d_{\overline{Y}}]) = e_2(\overline{Y}[d_{\overline{Y}}]),\ \forall d_{\overline{Y}} \in {}_{\mathcal{B}_1}U_{\overline{Y}} = {}_{\mathcal{B}_2}U_{\overline{Y}}$.

*Case 2 [$d \in {}_{\mathcal{D}_1}D_Z$]:* Here,

$$\begin{aligned}
Z[d] &= e_1(\overline{V}[d_{\overline{V}}]),\ \text{where } d_{\overline{V}} = {}_{\mathcal{D}_1}M_{Z,\overline{V}}(d) = {}_{\mathcal{D}'_1}M_{Z,\overline{V}}(d) \in {}_{\mathcal{D}_1}U_{\overline{V}}, \\
&\quad \text{since by construction of the directed acyclic version } \mathcal{D}'_1 \text{ of } \mathcal{D}_1, \\
&\quad {}_{\mathcal{D}_1}M_{Z,\overline{V}} = {}_{\mathcal{D}'_1}M_{Z,\overline{V}} \text{ over the domain } {}_{\mathcal{D}_1}D_Z = {}_{\mathcal{D}'_1}D_Z \\
&= r_{\mathcal{D}'_1}(\overline{V}[d_{\overline{V}}], Z1[d_{Z1}]),\ \text{for some } d_{Z1} \in {}_{\mathcal{D}'_1}U_{Z1} \quad \ldots(i)
\end{aligned}$$

However, ${}_{\mathcal{D}'_1}U_{Z1}(= {}_{\mathcal{D}_1}U_{Z1}) \subseteq {}_{\mathcal{B}_1}D_Z \bigcup {}_{\mathcal{D}'_1}D_Z$ (owing to recurrence). Because of SA property of the program(s), ${}_{\mathcal{B}_1}D_Z \bigcap {}_{\mathcal{D}'_1}D_Z = \emptyset$. Hence we have the following two (mutually exclusive) subcases: (2.1) $d_{Z1} \in {}_{\mathcal{B}_1}D_Z$ and (2.2) $d_{Z1} \in {}_{\mathcal{D}'_1}D_Z$.

*Subcase 2.1 [$d_{Z1} \in {}_{\mathcal{B}_1}D_Z$]:* So, continuing from (i) we have,

$$\begin{aligned}
Z[d] &= e_1(\overline{V}[d_{\overline{V}}]) \\
&= r_{\mathcal{D}'_1}(\overline{V}[d_{\overline{V}}], Z1[d_{Z1}]) \\
&= r_{\mathcal{D}'_1}(\overline{V}[d_{\overline{V}}], r_{\mathcal{B}_1}(\overline{Y}[d_{\overline{Y}}])) \\
&= r_{\mathcal{D}'_2}(\overline{V}[d_{\overline{V}}], r_{\mathcal{B}_2}(\overline{Y}[d_{\overline{Y}}])) \\
&= e_2(\overline{V}[d_{\overline{V}}]),\ \text{for some } d_{\overline{V}} = {}_{\mathcal{B}_1}M_{Z,\overline{Y}}(d) = {}_{\mathcal{B}_2}M_{Z,\overline{Y}}(d) \\
&\quad \text{since the equivalence checker ADDG\_EQX11 finds } {}_{\mathcal{B}_1}M_{Z,\overline{Y}}(d) = {}_{\mathcal{B}_2}M_{Z,\overline{Y}}(d), \\
&\quad r_{\mathcal{B}_1} = r_{\mathcal{B}_2} \text{ over } \{\overline{Y}[d_{\overline{Y}}], d_{\overline{Y}} \in {}_{\mathcal{B}_1}M_{Z,\overline{Y}}({}_{\mathcal{B}_1}D_Z) = {}_{\mathcal{B}_2}M_{Z,\overline{Y}}({}_{\mathcal{B}_2}D_Z)\} \text{ and} \\
&\quad r_{\mathcal{D}'_1} = r_{\mathcal{D}'_2} \text{ over } \{\overline{V}[d_{\overline{V}}], d_{\overline{V}} \in {}_{\mathcal{D}_1}U_{\overline{V}} = {}_{\mathcal{D}'_1}U_{\overline{V}}\} \times \{Z1[d_{z1}], d_{z1} \in {}_{\mathcal{D}'_1}U_{Z1}\}
\end{aligned}$$

Therefore, $e_1 = e_2$ for this subcase.

*Subcase 2.2 [$d_{Z1} \in {}_{\mathcal{D}'_1}D_Z$]:* It may be noted that from Definition 33 (recurrence array vertex) ${}_{\mathcal{D}_1}D_Z \bigcap {}_{\mathcal{D}_1}U_{Z1} = {}_{\mathcal{D}'_1}D_Z \bigcap {}_{\mathcal{D}'_1}U_{Z1} \neq \emptyset$, i.e., some elements of $Z$ defined through $\mathcal{D}_1$ are used in defining further elements of $Z$ in $\mathcal{D}_1$ itself. Specifically, equation (i) $Z[d] = e_1(\overline{V}[d_{\overline{V}}]) = r_{\mathcal{D}'_1}(\overline{V}[d_{\overline{V}}], Z1[d_{Z1}])$ depicts that in order to prove that $Z[d]$ evaluated through $e_1$ is the same as that evaluated through $e_2$, we need to *assume* that in $\mathcal{D}'_1$, the evaluation of $Z[d_{Z1}]$ $(= Z1[d_{Z1}])$ should precede the evaluation of $Z[d]$, and likewise, in $\mathcal{D}'_2$. Such an assumption is nothing but the induction hypothesis. (This aspect justifies the nomenclature of the subgraphs $\mathcal{D}_1$ and $\mathcal{D}_2$ as the induction subgraphs because it supports the inductive step in the analysis of a recurrence.) In other words, an ordering over the elements of ${}_{\mathcal{D}'_1}D_Z$ $(= {}_{\mathcal{D}'_2}D_Z)$ is needed for validation of $e_1 = e_2$. Towards this, let us consider the following relation: $\forall d_1, d_2 \in {}_{\mathcal{D}'_1}D_Z$ $(= {}_{\mathcal{D}'_2}D_Z)$, $d_1 \prec d_2$, if the computation of the value of $Z[d_2]$ depends upon the computation of that of $Z[d_1]$[2].

Assuming that $\forall d' \prec d$, $Z[d']$ $(= Z1[d'])$ are evaluated identically by $\mathcal{D}'_1$ and $\mathcal{D}'_2$, i.e.,

[2]Such a definition essentially implies that there is a *valid schedule*, which is supported by the restrictions given in Section 6.2.

$Z[d'] = e_1(\overline{V}[d'_{\overline{V}}]) = e_2(\overline{V}[d'_{\overline{V}}])$, we have

$$\begin{aligned} Z[d] &= e_1(\overline{V}[d_{\overline{V}}]) \\ &= r_{\mathcal{D}'_1}(\overline{V}[d_{\overline{V}}], Z1[d_{Z1}]) \\ &= r_{\mathcal{D}'_1}(\overline{V}[d_{\overline{V}}], e_1(\overline{V}[d'_{\overline{V}}])),\ \text{where } d'_{\overline{V}} = {}_{\mathcal{D}_1}M_{Z,\overline{V}}(d) = {}_{\mathcal{D}_2}M_{Z,\overline{V}}(d) \\ &= r_{\mathcal{D}'_2}(\overline{V}[d'_{\overline{V}}], e_2(\overline{V}[d'_{\overline{V}}])) \\ &= e_2(\overline{V}[d'_{\overline{V}}]),\ \text{by induction hypothesis and by the equivalence} \\ &\quad\ \text{checker ADDG\_EQX11 applied on } \mathcal{D}'_1 \text{ and } \mathcal{D}'_2 \end{aligned}$$

Hence, $e_1 = e_2$ over the defined domain ${}_{\mathcal{D}'_1}D_{Z1} =_{\mathcal{D}'_2} D_{Z1}$. ■

### 6.5.2 Complexity

Let us determine the worst case time complexity of all the steps involved in Algorithm 11. Note that the analysis has been done assuming the number of arrays in each ADDG is $a$, the number of statements (write edges) is $s$, the maximum arity of a function is $t$ and the number of recurrence subgraphs is $\gamma$.

*Step 1:* Finding the set of common arrays takes $a^2$ time and copying each ADDG takes $O(V+E)$, i.e., $O((a+s)+(a+s\times t))$ time.

*Step 2:* This step basically involves identification of SCCs in a directed graph using Tarjan's algorithm [148], which can be done in a depth-first traversal of the graph. Therefore, this step also requires $O(V+E) = O((a+s)+(a+s\times t))$ time.

*Steps 3—5:* Each of the three functions mentioned in step 4 requires a depth-first traversal of the ADDG; therefore, the loop takes $O(\gamma\times((a+s)+(a+s\times t)))$ time.

*Steps 6—8:* Similar to the steps 3—5.

*Steps 9—19:* In step 11, checking whether the arrays involved in two recurrence subgraphs are identical takes $O(a^2)$ time. Note that the complexity of comparing two normalized expressions is $O(2^{\|F\|})$, where $\|F\|$ is the length of a normalized formula in terms of its constituent variables and operators [88]. However, representing the data transformation of the elements of the output array of a subgraph in terms of the elements of the input arrays of that subgraph may require substitution of the intermediate temporary arrays in a transitive fashion, which takes $O(a^2\times 2^{\|F\|})$ time [88]. For finding the transitive dependence and the union of the mappings, ISL [152] has been used, whose worst case time complexity is the same as the deterministic upper bound on the time required to verify Presburger formulas, i.e., $O(2^{2^{2^n}})$, where $n = O(t \times a)$ is the length of the formula; the worst case behaviour, however, is never exhibited in our experiments. Since we replace the recurrence subgraphs (except for the recurrence vertex and the input array vertices) with an operator vertex representing an uninterpreted function along with the edges connecting this operator vertex with the recurrence vertex and the input array vertices, step 12 takes $O(V) = O(a+s)$ time for each ADDG. We have to compare the data transformations of pairs of recurrence subgraphs, one from each ADDG; thus, the loop encompassing the steps 9–19 takes $O(g \times (2^{\|F\|} + a^2 \times 2^{\|F\|}))$ time.

*Steps 20—22:* This step simply reports a failure case; however, in the worst case, the faulty recurrence subgraph may cover almost the entire ADDG; thus, this step may also require $O(V+E) = O((a+s) \times (a+s \times t))$ time.

*Steps 23—27:* In step 23, the verifier of [88] is invoked; therefore, the time complexity of these steps is $O(k^{n \times x} \times (2^{\|F\|} + a^2 \times 2^{\|F\|}) + k^{2 \times n \times x} \times 2^{\|F\|})$ in the worst case as reported in [88], where $n$ is the number of branching blocks in the control flow graph of the program, $k$ is the maximum branches in a branching block and $x$ is the maximum number of arrays defined in a branch. Since this is the costliest step in the entire algorithm, the worst case time complexity of Algorithm 11 is identical to the one reported in [88].

## 6.6 Experimental results

Our method has been implemented in the C language and run on a 2.0 GHz Intel® Core™2 Duo machine. The tool is available at http://cse.iitkgp.ac.in/~chitta/pubs/EquivalenceChecker_ADDG.tar.gz along with the benchmarks, installation and usage guidelines. It first constructs the ADDGs from the original and the transformed behaviours written in C and then applies our method to establish the equivalence between them. The tool has been tested on some benchmarks. The characteristics of the benchmarks and the time taken to establish equivalence by our tool and by those of [154] and [88] are given in Table 6.1; note that the symbol $\times$ has been used whenever a tool failed to establish the required equivalence. The benchmarks have been obtained by manually applying different transformations to the programs of Sobel edge detection (SOB), Debaucles 4-coefficient wavelet filter (WAVE), Laplace algorithm to edge enhancement of northerly directional edges (LAP), linear recurrence solver

Table 6.1: Results on some benchmarks

| Sl | | C lines | | loops | | arrays | | slices | | Exec time | Exec time | Exec time |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| No | Benchmark | src | trans | src | trans | src | trans | src | trans | (sec) [154] | (sec) [88] | (sec) [Our] |
| 1 | ACR1 | 14 | 20 | 1 | 3 | 6 | 6 | 1 | 1 | 0.18 | 0.76 | 0.28 |
| 2 | LAP1 | 12 | 28 | 1 | 4 | 2 | 4 | 1 | 2 | 0.28 | 9.25 | 0.55 |
| 3 | LIN1 | 13 | 13 | 3 | 3 | 4 | 4 | 2 | 2 | 0.12 | 0.62 | 0.24 |
| 4 | LIN2 | 13 | 16 | 3 | 4 | 4 | 4 | 2 | 3 | 0.13 | 0.74 | 0.30 |
| 5 | SOR | 26 | 22 | 8 | 6 | 11 | 11 | 1 | 1 | 0.18 | 1.08 | 0.68 |
| 6 | WAVE | 17 | 17 | 1 | 2 | 2 | 2 | 4 | 4 | 0.31 | 6.83 | 0.53 |
| 7 | ACR2 | 24 | 14 | 4 | 1 | 6 | 6 | 2 | 1 | × | 0.98 | 0.36 |
| 8 | LAP2 | 12 | 21 | 1 | 3 | 2 | 4 | 1 | 1 | × | 2.79 | 0.35 |
| 9 | LAP3 | 12 | 14 | 1 | 1 | 2 | 2 | 1 | 2 | × | 4.82 | 0.56 |
| 10 | LOWP | 13 | 28 | 2 | 8 | 2 | 4 | 1 | 2 | × | 9.17 | 0.63 |
| 11 | SOB1 | 27 | 19 | 3 | 1 | 4 | 4 | 1 | 1 | × | 1.79 | 0.61 |
| 12 | SOB2 | 27 | 27 | 3 | 3 | 4 | 4 | 1 | 1 | × | 1.85 | 0.57 |
| 13 | EXM1 | 8 | 14 | 1 | 1 | 2 | 4 | 1 | 3 | 0.14 | × | 0.36 |
| 14 | EXM2 | 8 | 15 | 1 | 1 | 3 | 5 | 1 | 3 | 0.19 | × | 0.48 |
| 15 | SUM1 | 8 | 14 | 1 | 1 | 2 | 4 | 1 | 3 | × | × | 0.40 |
| 16 | SUM2 | 16 | 19 | 4 | 4 | 4 | 6 | 2 | 4 | × | × | 0.72 |
| 17 | MUTR | 12 | 12 | 2 | 2 | 4 | 4 | 3 | 3 | × | × | 0.62 |

(LIN), successive over-relaxation (SOR), computation across (ACR), low-pass filter (LOWP), modified versions of the example given in Figure 6.6 (EXM), summation of the elements of different input arrays (SUM) and an example of mutual recursion (MUTR). Note that all the tools succeeded in showing equivalence for benchmarks 1–6 which involved only loop transformations; the tool of [154] failed to establish equivalence for benchmarks 7–12 because they contained arithmetic transformations as well; the tool of [88] failed for benchmarks 13–14 because they involved loop transformations along with recurrences and both the tools of [154] and [88] failed for benchmarks 15–17 because they involved both arithmetic transformations and recurrence; only our tool succeeded in showing equivalence in all the cases. Although a comparative analysis with the method of [144] would have also been relevant, we could not furnish it since their tool is not available to us. To find out the set of loop transformations and arithmetic transformations supported by our tool, the readers are referred to [88]. A pertinent point to note is that although our tool outperforms that of [88] with respect to execution time whenever both the tools are able to establish equivalence, the tool of [154] takes about 2.5 times less execution time than that of ours whenever it is successful – this is probably because our tool invokes ISL [152] through system call and communicates with it via reading and writing to files whereas, ISL comes as an integrated package within [154] itself and hence it is faster.

## 6.7 Conclusion

Loop and arithmetic transformations are applied extensively in embedded system design to minimize execution time, power, area, etc. An ADDG based equivalence checking method has been proposed in [86, 88] for translation validation of programs undergoing such transformations. This method, however, cannot be applied to verify programs that involve recurrences because recurrences lead to cycles in the ADDGs which render currently available dependence analyses and arithmetic simplification rules inadmissible. Another verification technique [153, 154] which can verify programs with recurrences does not handle arithmetic transformations. The validation scheme proposed in this chapter isolates the suitable subgraphs (arising from recurrences) in the ADDGs from the acyclic portions and treats them separately; each cyclic subgraph in the original ADDG is compared with its corresponding subgraph in the transformed ADDG in isolation and if all such pairs of subgraphs are found equivalent, then the entire ADDGs, with the subgraphs replaced by designated uninterpreted functions of proper arities, are compared in the usual manner of [88]. The soundness and the complexity of the method have been formally treated. The experimental results demonstrate the efficacy of the method. Our method, however, cannot resolve equivalence in the presence of recurrences that employ reductions [75] because reductions involve accumulations of a parametric number of sub-expressions using an associative and commutative operator. In our future work, we intend to alleviate the current limitation of the method to handle a more general class of programs.

# Chapter 7

# Conclusion and Scope for Future Work

Application of faulty code optimizations may proliferate as software bugs; hence it is crucial to perform translation validation. To meet this objective, some equivalence checking techniques have been developed and implemented by us to verify several code optimizations. In this chapter, we first briefly summarize the contributions of the work presented in the thesis. Next, we discuss several possible directions/extensions for future research.

## 7.1 Summary of contributions

**Translation validation of code motion transformations:** Code motion transformations are widely used to improve the performance of programs [48, 65, 66, 82]. Consequently, a lot of investigation has been devoted for verifying such transformations [95, 100, 104, 110]. Of the different techniques available for verification of code motion transformations, the path extension based technique [90, 91, 95, 110] which models programs as FSMDs [58] has been particularly encouraging since it has been successful in verifying code motion transformations in the presence of non-structure preserving transformations, i.e., transformations which alter control structures of programs, such as those introduced by path based schedulers [33, 135]. Moreover, this technique handles a wide range of arithmetic transformations by employing a normalization technique [141] that tries to reduce two computationally equivalent expressions to a syntactically identical form. The path extension based approach, however, fails in case of verifying code motions across loops because a path, by definition, cannot be extended beyond a loop [56]. Hence, our initial objective was to develop an equivalence checking method that alleviates this drawback of the path extension based approach while retaining all its benefits. We have developed a symbolic value propagation based equivalence checking method for FSMDs that stores the difference between the data computations of a pair of paths (obtained from two different FSMDs whose equivalence we seek to validate) as propagated vectors and propagates these vectors to all the paths originating from the end states of the original path pair. This process is repeated until all the mismatches encountered have been compensated for during subsequent traversal of the FSMDs; if no mismatch is identified when the final paths of the two FSMDs have been traversed, then we declare them as equivalent; otherwise, we rule them as *possibly* non-equivalent. Note that equivalence checking of flowchart schemas being an undecidable problem, completeness cannot be assured. However, the soundness and the termination of our method have been formally proved. To examine whether our symbolic value propagation based equivalence checking is superior to path extension based equivalence checking, we initially compared our tool with that of [95] for three different compilers namely, a synthesis tool for structured datapaths [117], a path based scheduler [33] and a high-level synthesis tool SPARK [64]. For all three compilers, our tool required less time on an average to establish equivalence than that of [95]. Next, we compared our tool with that of [90], which is an improvised method of [95], that can additionally handle non-uniform code motions; our tool was found to take considerably less amount of time in establishing equivalence for all the benchmarks reported in [90]. Finally, we subjected some benchmarks to SPARK followed by the path based scheduler to perform optimizations in two successive steps; for those test cases where the target code was produced from the source code by means of code motions across loops, only our tool could prove the equivalence. It is important to note that the computational complexity of the symbolic value propagation based method has been analyzed and found to be no worse than that for path extension method [90], i.e., our symbolic value propagation method is capable of handling more sophisticated transformations than [90] without incurring any extra overhead of time complexity.

**Deriving bisimulation relations from path based equivalence checkers:** Translation validation by means of deriving bisimulation relations between programs has been an actively persuaded field of study [103, 104, 126]. Meanwhile, translation validation through path based equivalence checking has also received similar attention [22, 90, 95, 110]. Both these techniques have been popular because they provide benefits which are exclusive to each, while suffering from distinctive drawbacks. Specifically, bisimulation based methods [103, 104] can handle loop shifting [44], which no path based approach can handle till date, whereas, path based approaches [22, 90, 95, 110] are adept in handling non-structure preserving transformations; furthermore, path based approach guarantees termination which bisimulation based approach cannot. However, the conventionality of bisimulation as the approach for equivalence checking raises the natural question of examining whether path based equivalence checking yields a bisimulation relation or not. In this thesis, we have presented mechanisms to derive bisimulation relations from the outputs of the path extension based equivalence checker and the symbolic value propagation based equivalence checker whenever two FSMDs are found to be equivalent by these checkers. It is to be noted that none of the bisimulation relation based approaches has been shown to tackle code motions across loops; therefore, the present work demonstrates, for the first time, that a bisimulation relation exists even under such a context.

**Translation validation of code motion transformations in array-intensive programs:** The FSMD model does not support arrays. As a result, the verifiers built upon this model are rendered inapplicable while verifying code motion transformations in array-intensive programs. To alleviate this shortcoming, several ramifications are needed. First, the FSMD model is extended to the FSMDA model which allows arrays in its datapath. Secondly, the normalization technique of [141] is augmented with additional grammar rules to represent array references as McCarthy's functions [120]. Thirdly, the symbolic value propagation based equivalence checker is fitted with the FSMDA model while accommodating special propagation rules for index variables. The correctness and the complexity of the verification procedure are formally treated. To validate that our FSMDA based equivalence checker is capable of verifying code motion transformations of array-intensive programs, we performed a set of experiments. The benchmark suite comprised behaviours which are computation intensive and primarily involved arrays. These benchmarks were fed to the SPARK tool [64] to obtain the transformed behaviours. New scalar variables, but no array variables, are introduced during the transformations. Our proposed method could ascertain equivalences in all the cases except one; for this case, it reported a possible non-equivalence and outputted a set of paths (each paired with an almost similar path) for which equivalence could not be found. A scrutiny of the involved benchmark revealed that the non-equivalence resulted from a bug in the implementation of copy propagation for array variables in the SPARK tool.

**Translation validation of loop and arithmetic transformations in the presence of recurrences:** Loop and arithmetic transformations are applied extensively in embedded system design to minimize execution time, power, area, etc. [30, 79, 131, 165]. An equivalence checking method which models programs as ADDGs has been proposed in [144, 146] for translation validation of programs undergoing loop transformations. This method is extended later in [86, 88] to verify a plethora of arithmetic transformations as well. This method, however, cannot be applied to verify programs that involve recurrences because recurrences lead to cycles in the ADDGs which render currently available dependence analyses and arithmetic simplification rules inadmissible. Another verification technique [154] which can verify programs with recurrences does not handle arithmetic transformations. Therefore, we aimed at developing a unified equivalence checking framework to cover the entire spectrum of loop and arithmetic transformations in addition to recurrences by further extending the ADDG based equivalence checking strategy. Our method initially segregates the subgraphs representing recurrences in an ADDG from the cycle-free subgraphs and basically tries to find pairs of such recurrence subgraphs, one from the original ADDG and the other from the transformed ADDG, which have undergone equivalent functional transformation; on finding a match, it substitutes the recurrence subgraphs in either ADDG with an identical uninterpreted function; once all recurrence subgraphs in an ADDG have been successfully paired up with recurrence subgraphs from the other ADDG and subsequently substituted by uninterpreted functions, the equivalence checker of [88] is applied on the modified ADDGs (which no longer contain cycles) to test their equivalence. The correctness of the proposed method is formally proved and the complexity is analyzed. The experimental results attest the effectiveness of the method; while the method of [154] is able to establish equivalence in cases where the transformed behaviour is obtained from the original by application of loop transformations and recurrences (but no arithmetic transformation), the method of [88] can prove equivalence in cases of loop and arithmetic transformations (but no recurrence); only our method can establish equivalence is presence of loop and arithmetic transformations along with recurrences.

It is to be noted that our tools along with the benchmarks and usage guidelines are available for download at http://cse.iitkgp.ac.in/~chitta/pubs/ and are annotated as "Assorted formal equivalence checking programs."

## 7.2 Scope for future work

During the course of the thesis work, we had several interesting realizations regarding how to overcome the current limitations and how to apply the developed procedures in other related fields. In the following, we discuss both aspects of future works.

### 7.2.1 Enhancement of the present work

**Allowing arrays to appear in the subscript of other arrays:** Our current symbolic value propagation based equivalence checker for FSMDAs does not support programs which have arrays containing other arrays in their subscripts. A possible remedy is to allow such constructs and mark the *index array variables*, similar to the current index variables which presently comprise scalar variables only. The rule of propagating values of index scalar variables in spite of a match also has to be extended to the index array variables. However, such a remedy will obviously entail more bookkeeping to store all these index variables.

```
A[0] = in;
for ( i = 1; i <= N; ++i ) {
 A[i] = f(g(A[i/2]));
}
out = g(A[N]);
```

(a) Original program.

```
A[0] = g(in);
for ( i = 1; i <= N; ++i ) {
 A[i] = g(f(A[i/2]));
}
out = A[N];
```

(b) Transformed program.

Figure 7.1: A pair of equivalent programs with non-equivalent recurrence graphs.

**Combining symbolic value propagation with ADDG based equivalence checking method:** Figure 7.1 shows a pair of equivalent programs with non-equivalent recurrence subgraphs. Since the basis subgraphs and the induction subgraphs of the programs are different, our ADDG based equivalence checker will declare them to be possibly non-equivalent. However, an equivalence checking mechanism may be devised that propagates the mismatched symbolic values *in* and $g(in)$ from the basis subgraphs to the induction subgraphs and even out of the recurrence subgraph, if required (as in the present scenario). Note that the method of [154] is able to establish equivalence of the two programs given in Figure 7.1 by employing a similar technique.

**Attending currently unaddressed programming constructs:** Here we present some of the ways in which our present equivalence checking frameworks can be extended to encompass currently unaddressed programming constructs.

*Unsupported datatypes:* The normalization technique employed by our path based equivalence checkers reduces many computationally equivalent expressions to a syntactically identical form. In order to compare the path characteristics, their conditions of execution and data transformations are represented in this normalized form. However, the present normalization grammar has no rules to represent bit-vectors and user-defined datatypes. An SMT solver can be used to overcome these drawbacks. This problem has been explored by us in [23] where we experimented with three SMT solvers – Yices2, CVC4 and Z3. A pertinent point to note is that whenever the normalization technique was able to establish equivalence its execution time was found to be less than those involving SMT solvers. This observation indicates that reducing two expressions to identical structural form (using normalization) to determine their equivalence is a less time-consuming process than applying algebraic simplification rules (such as those employed by the SMT solvers reported in this work) for the same. So, we think it is better to invoke an SMT solver only when the normalizer is found to be inadequate in establishing equivalence between two expressions, i.e., use an SMT solver to supplement the normalizer and not to replace it. A full fledged implementation of such an amalgamated equivalence checker still remains as a future work.

*Pointers and dynamic memory allocation:* We envision that posterior to alias analysis if pointers are replaced by the actual storage variables that they point to, such as, $*(p+i)$ is converted to $p[i]$ where $p$ is the base address of an array and $i$ represents an index, then our equivalence checking procedures can be applied as usual. For dynamic memory allocation, fragmentation and garbage collection presents further complicacies. The Memcheck tool in the Valgrind tool suite [8] addresses many of the invalid memory access issues. Once the validity of the memory accesses are ascertained then our tools may be used.

*Inter-procedural analysis:* The equivalence checking procedures described in this thesis compare one function from the original program with one from the transformed program. In the scenario where we want to check equivalence between an original and a transformed program comprising multiple functions, we may do so by checking equivalence between pairs of programs (one from the original program and the other from the transformed program) in a bottom-up fashion, i.e., we start with the functions which which have no dependency on any of the other functions before moving on to those which have dependencies on the functions which are already proved to be equivalent. Note that the functions can be paired easily based on their names because typically compilers do not rename functions. In case our equivalence checker is employed to verify semi-automated or hand-optimized programs where some of the functions may have been split or merged in the transformed version, one may inline the functions within one *main* function and then apply our equivalence checker from program verification. Of course, inlining may not be viable always, especially in the presence of recursive functions, and in such a case our method may not be applicable.

*Concurrency:* In a scenario where thread-safety guarantees that the code is free of race conditions, i.e., the final values stored can be accurately computed, our equivalence checking technique may be applied to check equivalence between the original sequential code and the transformed parallel code where only one schedule of the threads is considered. Otherwise, path based equivalence checking methods using Petri net based models, which can effectively capture concurrency, may be used [18, 19].

### 7.2.2 Scope of application to other research areas

**Evolving programs:** Although the present thesis deals with validation of code optimizations that are commonly applied by compilers, the scope of checking equivalence between programs may include other domains as well where the transformed programs have been obtained by some other means. One such domain is *evolving programs*; software is not written entirely from scratch, it rather evolves over time. Often industrial software development projects release different versions of the same software; validation of such evolving programs to ensure compatibility between these versions remains a huge problem [133]. An interesting study would be to check the applicability of the formal methods developed in this thesis to establish equivalence of evolving programs.

```
C = 0;
A[3:0] = 0;
X[3:0] = 0;
M[3:0] = multiplicand[3:0];
Q[3:0] = multiplier[3:0];
count[2:0] = 100;
do
begin
 if(Q[0] == 1)
  {C, A[3:0]} = A[3:0] + M[3:0];
 end
 Q[3:0] = Q[3:0] >> 1;
 {C, A[3:0], X[3:0]} =
   {C, A[3:0], X[3:0]} >> 1;
 count[2:0] = count[2:0] - 1;
end
while(count != 0);
prod[7:0] = {A[3:0], X[3:0]};
```

(a) Shift-add multiplication with right shifting (C, A, Q).

```
A[3:0] = 0;
M[3:0] = multiplicand[3:0];
Q[3:0] = multiplier[3:0];
count[2:0] = 100;
do
begin
 if(Q[0] == 1)
  Q[3:0] = Q[3:0] >> 1;
  {A[3:0], Q[3]} = A[3:0] + M[3:0];
 else
  Q[3:0] = Q[3:0] >> 1;
  {A[3:0], Q[3]} = A[3:0] + 0;
 end
 count[2:0] = count[2:0] - 1;
end
while(count != 0);
prod[7:0] = {A[3:0], Q[3:0]};
```

(b) Shift-add multiplication with right shifting only Q.

Figure 7.2: A pair of programs with shift-add multiplication.

**Checking equivalence at bit-level:** Our equivalence checkers verify equivalence of arithmetic transformations by employing a normalization technique [141], that has been extended by us later in [25], which tries to reduce two computationally equivalent expressions to a syntactically identical form. This normalization technique, however, considers integers to be of arbitrary size, i.e., finite size integers are not handled; moreover, it does not have provision for bitwise operators. Hence, our techniques cannot be readily applied to verify programs involving bit-vectors, e.g., the two equivalent versions of shift-add multiplication algorithm coded in Verilog given in Figure 7.2. It is to be noted that some work has been done by our research group to solve this problem [137], which adopts our notions of path covers and symbolic value propagation; however, many challenges still remain unexplored.

**Automatic program evaluation:** Manual assessment of student programs is often slow and inconsistent. Usually, in the introductory programming courses offered in institutions with a large intake of students, a lot of assessment work has to be done manually by the instructors. As an example, at IIT Kharagpur where introductory programming course is offered to around 600 students in every semester, each student has to submit around 10–12 assignments and each assessment has about 10 programming exercises. Thus, the evaluation task becomes as enormous as checking around 1,20,000 programs in a year. Assessment speed can be improved along with consistency by automating the process of evaluation. Hence, it is desirable to develop tools for automated evaluation of programming assignments. A survey on automated assessment of programs can be found in [11]. An automated program evaluation scheme by leveraging the equivalence checking method of FSMDs is presented in [143], where a student program is compared with the program supplied by the instructor, both modeled as FSMDs. However, a lot more headway is still required for this problem to reach a full-fledged deployment.

# Appendix A

# Construction of FSMDAs from Behavioural Descriptions

This appendix gives a brief elucidation for obtaining FSMDAs from behavioural descriptions (high-level languages), such as C, and how to represent them textually so that they can be fed as inputs to our FSMDA equivalence checker. Note that the content of section A.1 of this manuscript is largely borrowed from [90].

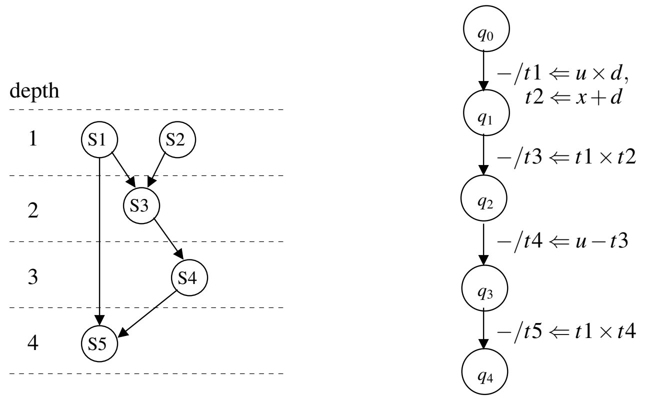
## A.1 How to represent behavioural descriptions as FSMDAs conceptually

Any (sequential) behaviour consists of a combination of the following three basic constructs: (i) sequences of statements without any bifurcation of control flow, i.e., Basic Blocks (BBs), (ii) if-else constructs, i.e., Control Blocks (CBs), and (iii) loops. Therefore, without any loss of generality, capturing these three constructs in an FSMDA model enables us to effectively represent any sequential behaviour as an FSMDA.

A BB consisting of a sequence of $n$ statements $S_1, \ldots, S_n$, can be represented as a sequence of $n+1$ states $q_0, \ldots, q_n$, say and $n$ edges of the form $q_{j-1} \xrightarrow{-/S_j} q_j, 1 \leq j \leq n$, in the corresponding FSMDA. The number of states in the FSMDA, however, can be reduced in the following way (although it is not mandatory).

S1: $t1 \Leftarrow u \times d$;
S2: $t2 \Leftarrow x + d$;
S3: $t3 \Leftarrow t1 \times t2$;
S4: $t4 \Leftarrow u - t3$;
S5: $t5 \Leftarrow t1 \times t4$;

(a) A sequence of statements.

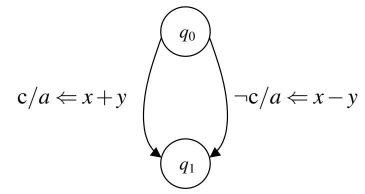

depth
1
2
3
4
S1
S2
S3
S4
S5
$q_0$
$-/t1 \Leftarrow u \times d,$
$t2 \Leftarrow x + d$
$q_1$
$-/t3 \Leftarrow t1 \times t2$
$q_2$
$-/t4 \Leftarrow u - t3$
$q_3$
$-/t5 \Leftarrow t1 \times t4$
$q_4$

(b) Its dependence graph. (c) Corresponding FSMDA.

Figure A.1: Construction of FSMDA corresponding to a basic block.

```
if ( c )
  a ⇐ x + y;
else
  a ⇐ x − y;
endif
```

(a) An if-else block.



(b) Corresponding FSMDA.

Figure A.2: Construction of FSMDA corresponding to a control block.

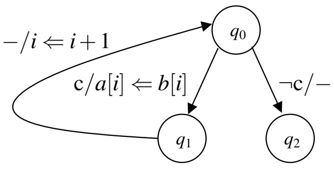
For a BB, we first construct a dependence graph. The graph consists of a node corresponding to each statement of the BB. There is a directed edge in the dependence graph from the statement $S1$ to the statement $S2$ iff there is one of *read-after-write*, *write-after-read*, and *write-after-write* dependences between $S1$ and $S2$. Naturally, the constructed graph is a directed acyclic graph. Let the height of the dependence graph be $h$. We now construct an FSMDA of $h+1$ states $q_0, \ldots, q_h$, say and $h$ edges of the form $q_j \twoheadrightarrow q_{j+1}, 0 \leq j \leq h-1$. We place the operations at depth $k, 1 \leq k \leq h$, of the dependence graph in the transition $q_{k-1} \twoheadrightarrow q_k$ of the FSMDA. The condition associated with each transition is $true$. For example, the dependence graph for the BB in Figure A.1(a) is depicted in Figure A.1(b) and the corresponding FSMDA is given in Figure A.1(c).

A CB is of the form: `if(c) then BB1 else BB2 endif`, where `c` is a conditional statement and `BB1` and `BB2` are two basic blocks which execute when `c` is $true$ and $false$, respectively. In this case, we construct the FSMDAs corresponding to `BB1` and `BB2` first. The FSMDA of the CB is obtained from these two FSMDAs by: (i) merging the start states of two FSMDAs into one start state and the end states of two

FSMDAs into one end state, and (ii) the condition `c` is placed as the condition of the first transition of the FSMDA corresponding to `BB1` and the condition ¬`c` is placed in the first transition of the FSMDA corresponding to `BB2`. A sample if-else block is shown in Figure A.2(a) and its corresponding FSMDA is shown in Figure A.2(b).

```
while ( c )
  $a[i] \Leftarrow b[i]$;
  $i \Leftarrow i+1$;
endwhile
```

(a) A while block.



(b) Corresponding FSMDA.

Figure A.3

For the loop constructs, let us consider only *while* loops without any loss of generality. A while loop is of the form `while(c) BB1 endwhile`. In this case, we first construct the FSMDA $M$ of `BB1`. The FSMDA corresponding to the while loop is obtained by merging the start and the end state of $M$ into one state, $q_0$ say, and placing the condition `c` in the transitions from $q_0$. We also need to add a transition from $q_0$ with condition ¬`c` as the exit path from the loop in the FSMDA as shown in Figure A.3.

## A.2 How to represent FSMDAs textually

Now that we have described how behavioural descriptions can be represented as FSMDAs, we explain how FSMDAs are represented textually so that they can be parsed by our equivalence checker.

The first line of the text file describing the FSMDA must contain its name (a string) within double quotes, for example, `"My_FSMDA"`.
Each transition in the FSMDA is represented using the following rule:

```
source_state number_of_transtions
  condition_1 | data_transformations_1 destination_state_1
[ condition_2 | data_transformations_2 destination_state_2 ] ;
```

Thus, the FSMDA shown in Figure A.1(c) is represented as follows:

```
q0 1 - | t1 = u * d, t2 = x + d  q1 ;
q1 1 - | t3 = t1 * t2            q2 ;
q2 1 - | t4 = u - t3             q3 ;
q3 1 - | t5 = t1 * t4            q4 ;
```

Note that the operations that occur in the same transition are separated by commas. Moreover, we also refrain from using special symbols, such as '⇐' and '×', which generally appear in our published papers since they are not readily available in standard keyboards; instead we use symbols, such as '=' and '*' for representing assignment and multiplication, respectively (we use '==' for checking equality, similar to C). Furthermore, we have also replaced '/' by '|' to separate the condition of execution and data transformation of a transition since the symbol '/' is also used to represent the division operation and therefore, introduced *reduce/reduce* conflict [2] during parsing.

The FSMDA shown in Figure A.2(b) is represented as given below:

```
q0 2  c | a = x + y  q1
     !c | a = x - y  q1 ;
```

While the FSMDA shown in Figure A.3(b) is represented as follows:

```
q0 2  c | a[i] = b[i]  q1
     !c | -            q2 ;
q1 1  - | i = i + 1    q0 ;
```

Thus, the symbol '!' is used (instead of ¬) to denote negation of a condition; the symbol '-' represents *true* when it appears as the condition of execution of a transition whereas, the same symbol represents *no operation* when it appears as the data transformation of a transition.

It is important to note that there can be any number of transitions occurring from a state in an FSMDA, see Figure 1 of reference [95] for example, as long as the condition of execution of each of the transitions is mutually exclusive from each of the other transitions.

The following operation depicts that the value read from port `P` is stored in the variable `v`:

```
read( v, P )
```

Similarly, the following operation depicts that the expression `e` is written into port `P`:

```
write( P, e )
```

The users are requested not to confuse these `read` and `write` operations with those of McCarthy's read/write operations [120] that are associated with arrays; the internal representation of array expressions using McCarthy's read/write operations by our equivalence checker is handled in a different manner than the `read` and `write` operations involving ports. Lastly, a final state (having a single transition, with *true* as its condition of execution and *no operation* as its data transformation, back to the reset state), `qL` say, in an FSMDA is represented as follows:

```
qL 0 ;
```

# Bibliography

[1] ACL2 Version 6.1. http://www.cs.utexas.edu/~moore/acl2/.

[2] Bison: Reduce/Reduce Conflicts. https://www.gnu.org/software/bison/manual/html_node/Reduce_002fReduce.html.

[3] CVC4 - the smt solver. http://cvc4.cs.nyu.edu/web/.

[4] GCC, the GNU Compiler Collection. https://gcc.gnu.org/.

[5] The llvm compiler infrastructure. http://llvm.org/.

[6] PVS Specification and Verification System. http://pvs.csl.sri.com/.

[7] The Yices SMT Solver. http://yices.csl.sri.com/.

[8] Valgrind. http://valgrind.org/.

[9] Z3. http://z3.codeplex.com/.

[10] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Princiles, Techniques, and Tools*. Pearson Education, 2006.

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

[12] R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures*. Morgan Kaufmann Publishers, 2001.

[13] D. F. Bacon, S. L. Graham, and O. J. Sharp. Compiler transformations for high-performance computing. *ACM Comput. Surv.*, 26:345–420, 1994.

[14] C. Baier. Polynomial time algorithms for testing probabilistic bisimulation and simulation. In *Computer Aided Verification*, pages 50–61, 1996.

[15] C. Baier, B. Engelen, and M. E. Majster-Cederbaum. Deciding bisimilarity and similarity for probabilistic processes. *J. Comput. Syst. Sci.*, 60(1):187–231, 2000.

[16] C. Baier and J.-P. Katoen. *Principles of model checking*. MIT Press, 2008.

[17] S. Bandyopadhyay, K. Banerjee, D. Sarkar, and C. Mandal. Translation validation for PRES+ models of parallel behaviours via an FSMD equivalence checker. In *Progress in VLSI Design and Test*, pages 69–78, 2012.

[18] S. Bandyopadhyay, D. Sarkar, K. Banerjee, and C. Mandal. A path-based equivalence checking method for petri net based models of programs. In *International Conference on Software Engineering and Applications,*, pages 319–329, 2015.

[19] S. Bandyopadhyay, D. Sarkar, and C. Mandal. An efficient equivalence checking method for petri net based models of programs. In *International Conference on Software Engineering*, pages 827–828, 2015.

[20] K. Banerjee. An equivalence checking mechanism for handling recurrences in array-intensive programs. In *Principles of Programming Languages: Student Research Competition*, pages 1–2, 2015.

[21] K. Banerjee, C. Karfa, D. Sarkar, and C. Mandal. A value propagation based equivalence checking method for verification of code motion techniques. In *International Symposium on Electronic System Design*, pages 67–71, 2012.

[22] K. Banerjee, C. Karfa, D. Sarkar, and C. Mandal. Verification of code motion techniques using value propagation. *IEEE Trans. on CAD of ICS*, 33(8):1180–1193, 2014.

[23] K. Banerjee, C. Mandal, and D. Sarkar. Extending the scope of translation validation by augmenting path based equivalence checkers with smt solvers. In *VLSI Design and Test, 18th International Symposium on*, pages 1–6, July 2014.

[24] K. Banerjee, C. Mandal, and D. Sarkar. Deriving bisimulation relations from path extension based equivalence checkers. In *WEPL*, pages 1–2, 2015.

[25] K. Banerjee, D. Sarkar, and C. Mandal. Extending the FSMD framework for validating code motions of array-handling programs. *IEEE Trans. on CAD of ICS*, 33(12):2015–2019, 2014.

[26] G. Barany and A. Krall. Optimal and heuristic global code motion for minimal spilling. In *Compiler Construction*, pages 21–40, 2013.

[27] C. W. Barrett, Y. Fang, B. Goldberg, Y. Hu, A. Pnueli, and L. D. Zuck. Tvoc: A translation validator for optimizing compilers. In *Computer Aided Verification*, pages 291–295, 2005.

[28] D. Barthou, P. Feautrier, and X. Redon. On the equivalence of two systems of affine recurrence equations (research note). In *Euro-Par Conference on Parallel Processing*, pages 309–313, 2002.

[29] S. Blazy and X. Leroy. Mechanized semantics for the Clight subset of the C language. *J. Autom. Reasoning*, 43(3):263–288, 2009.

[30] Y. Bouchebaba, B. Girodias, G. Nicolescu, E. M. Aboulhamid, B. Lavigueur, and P. G. Paulin. MPSoC memory optimization using program transformation. *ACM Trans. Design Autom. Electr. Syst.*, 12(4):43:1–43:39, 2007.

[31] F. Brandner and Q. Colombet. Elimination of parallel copies using code motion on data dependence graphs. *Computer Languages, Systems & Structures*, 39(1):25–47, 2013.

[32] C. Brandolese, W. Fornaciari, F. Salice, and D. Sciuto. Analysis and modeling of energy reducing source code transformations. In *Design, Automation and Test in Europe*, pages 306–311, 2004.

[33] R. Camposano. Path-based scheduling for synthesis. *IEEE Trans. on CAD of ICS*, 10(1):85–93, 1991.

[34] F. Catthoor, E. D., and S. S. Greff. *HICSS. Custom Memory Management Methodology: Exploration of Memory Organisation for Embedded Multimedia System Design.* Kluwer Academic Publishers, 1998.

[35] G. Chen, M. Kandemir, and F. Li. Energy-aware computation duplication for improving reliability in embedded chip multiprocessors. In *Asia and South Pacific Design Automation Conference*, pages 134–139, 2006.

[36] T.-H. Chiang and L.-R. Dung. Verification method of dataflow algorithms in high-level synthesis. *J. Syst. Softw.*, 80(8):1256–1270, 2007.

[37] A. Chutinan and B. H. Krogh. Verification of infinite-state dynamic systems using approximate quotient transition systems. *IEEE Trans. Automat. Contr.*, 46(9):1401–1410, 2001.

[38] A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri. Nusmv: a new symbolic model checker. *International Journal on Software Tools for Technology Transfer*, 2:410–425, 2000.

[39] J. Cong, P. Zhang, and Y. Zou. Combined loop transformation and hierarchy allocation for data reuse optimization. In *International Conference on Computer-Aided Design*, pages 185–192, 2011.

[40] J. Cong, P. Zhang, and Y. Zou. Optimizing memory hierarchy allocation with loop transformations for high-level synthesis. In *Design Automation Conference*, pages 1233–1238, 2012.

[41] R. Cordone, F. Ferrandi, M. D. Santambrogio, G. Palermo, and D. Sciuto. Using speculative computation and parallelizing techniques to improve scheduling of control based designs. In *Asia and South Pacific Design Automation Conference*, pages 898–904, 2006.

[42] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Second Edition*. The MIT Press and McGraw-Hill Book Company, 2001.

[43] L. A. Cortés, P. Eles, and Z. Peng. Verification of embedded systems using a petri net based representation. In *Proceedings of the 13th International Symposium on System Synthesis*, pages 149–156, 2000.

[44] A. Darte and G. Huard. Loop shifting for loop compaction. *J. Parallel Programming*, 28(5):499–534, 2000.

[45] R. De Nicola, A. Fantechi, S. Gnesi, and G. Ristori. An action based framework for verifying logical and behavioural properties of concurrent systems. In *Computer Aided Verification*, pages 37–47, 1991.

[46] P. C. Diniz and J. M. P. Cardoso. Code transformations for embedded reconfigurable computing architectures. In *Generative and Transformational Techniques in Software Engineering III*, pages 322–344, 2009.

[47] L. C. V. Dos Santos, M. J. M. Heijligers, C. A. J. Van Eijk, J. Van Eijnhoven, and J. A. G. Jess. A code-motion pruning technique for global scheduling. *ACM Trans. Des. Autom. Electron. Syst.*, 5(1):1–38, 2000.

[48] L. C. V. Dos. Santos and J. Jress. A reordering technique for efficient code motion. In *Design Automation Conference*, pages 296–299, 1999.

[49] A. Dovier, C. Piazza, and A. Policriti. An efficient algorithm for computing bisimulation equivalence. *Theor. Comput. Sci.*, 311(1-3):221–256, 2004.

[50] A. P. N. E. Özer and D. Gregg. Classification of compiler optimizations for high performance, small area and low power in fpgas. Technical report, Trinity College, Dublin, Ireland, Department of Computer Science, 2003.

[51] H. Eveking, H. Hinrichsen, and G. Ritter. Automatic verification of scheduling results in high-level synthesis. In *Design, Automation and Test in Europe*, pages 260–265, 1999.

[52] H. Falk and P. Marwedel. *Source code optimization techniques for data flow dominated embedded software*. Kluwer, 2004.

[53] J. Fernandez and L. Mounier. "on the fly" verification of behavioural equivalences and preorders. In *Computer Aided Verification*, pages 181–191, 1991.

[54] J. Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Transactions on Computers*, C-30(7):478–490, 1981.

[55] K. Fisler and M. Y. Vardi. Bisimulation minimization in an automata-theoretic verification framework. In *Formal Methods in Computer-Aided Design*, pages 115–132, 1998.

[56] R. W. Floyd. Assigning meaning to programs. In *Proceedings the 19th Symposium on Applied Mathematics*, pages 19–32, 1967.

[57] A. Fraboulet, K. Kodary, and A. Mignotte. Loop fusion for memory space optimization. In *International Symposium on Systems Synthesis*, pages 95–100, 2001.

[58] D. D. Gajski, N. D. Dutt, A. C. Wu, and S. Y. Lin. *High-Level Synthesis: Introduction to Chip and System Design*. Kluwer Academic, 1992.

[59] M. Ghodrat, T. Givargis, and A. Nicolau. Control flow optimization in loops using interval analysis. In *Proceedings of the 2008 international conference on Compilers, architectures and synthesis for embedded systems*, pages 157–166, 2008.

[60] M. Ghodrat, T. Givargis, and A. Nicolau. Optimizing control flow in loops using interval and dependence analysis. *Design Automation for Embedded Systems*, 13:193–221, 2009.

[61] R. Gupta and M. Soffa. Region scheduling: an approach for detecting and redistributing parallelism. *IEEE Transactions on Software Engineering*, 16(4):421–431, 1990.

[62] S. Gupta, N. Dutt, R. Gupta, and A. Nicolau. Dynamic conditional branch balancing during the high-level synthesis of control-intensive designs. In *Design, Automation and Test in Europe*, pages 270–275, 2003.

[63] S. Gupta, N. Dutt, R. Gupta, and A. Nicolau. Dynamically increasing the scope of code motions during the high-level synthesis of digital circuits. *IEE Proceedings: Computer and Digital Technique*, 150(5):330–337, 2003.

[64] S. Gupta, N. Dutt, R. Gupta, and A. Nicolau. SPARK: A high-level synthesis framework for applying parallelizing compiler transformations. In *VLSI Design*, pages 461–466, 2003.

[65] S. Gupta, N. Dutt, R. Gupta, and A. Nicolau. Using global code motions to improve the quality of results for high-level synthesis. *IEEE Trans. on CAD of ICS*, 23(2):302–312, Feb 2004.

[66] S. Gupta, R. Gupta, N. Dutt, and A. Nicolau. Coordinated parallelizing compiler optimizations and high-level synthesis. *ACM Trans on Design Automation of Electronic Systems*, 9(4):1–31, 2004.

[67] S. Gupta, R. K. Gupta, M. Miranda, and F. Catthoor. Analysis of high-level address code transformations for programmable processors. In *Design, Automation and Test in Europe*, pages 9–13, 2000.

[68] S. Gupta, M. Reshadi, N. Savoiu, N. Dutt, R. Gupta, and A. Nicolau. Dynamic common sub-expression elimination during scheduling in high-level synthesis. In *Proceedings of the 15th International Symposium on System Synthesis*, pages 261–266, 2002.

[69] S. Gupta, N. Savoiu, N. Dutt, R. Gupta, and A. Nicolau. Conditional speculation and its effects on performance and area for high-level synthesis. In *International Symposium on System Synthesis*, pages 171–176, 2001.

[70] S. Gupta, N. Savoiu, S. Kim, N. Dutt, R. Gupta, and A. Nicolau. Speculation techniques for high level synthesis of control intensive designs. In *Design Automation Conference*, pages 269–272, 2001.

[71] R. A. Hernandez, M. Strum, and W. J. Chau. Transformations on the FSMD of the RTL code with combinational logic statements for equivalence checking of HLS. In *Latin-American Test Symposium*, pages 1–6, 2015.

[72] Y. Hu, C. W. Barrett, B. Goldberg, and A. Pnueli. Validating more loop optimizations. *Electr. Notes Theor. Comput. Sci.*, 141(2):69–84, 2005.

[73] C. Hur, D. Dreyer, G. Neis, and V. Vafeiadis. The marriage of bisimulations and kripke logical relations. In *Principles of Programming Languages*, pages 59–72, 2012.

[74] W. M. W. Hwu, S. A. Mahlke, W. Y. Chen, P. P. Chang, N. J. Warter, R. A. Bringmann, R. G. Ouellette, R. E. Hank, T. Kiyohara, G. E. Haab, J. G. Holm, and D. M. Lavery. The superblock: An effective technique for vliw and superscalar compilation. *The Journal of Supercomputing*, 7:229–248, 1993.

[75] G. Iooss, C. Alias, and S. V. Rajopadhye. On program equivalence with reductions. In *Static Analysis*, pages 168–183, 2014.

[76] R. Jain, A. Majumdar, A. Sharma, and H. Wang. Empirical evaluation of some high-level synthesis scheduling heuristics. In *Design Automation Conference*, pages 686–689, 1991.

[77] N. E. Johnson. *Code Size Optimization for Embedded Processors*. PhD thesis, University of Cambridge, 2004.

[78] I. Kadayif and M. T. Kandemir. Data space-oriented tiling for enhancing locality. *ACM Trans. Embedded Comput. Syst.*, 4(2):388–414, 2005.

[79] I. Kadayif, M. T. Kandemir, G. Chen, O. Ozturk, M. Karaköy, and U. Sezer. Optimizing array-intensive applications for on-chip multiprocessors. *IEEE Trans. Parallel Distrib. Syst.*, 16(5):396–411, 2005.

[80] A. Kanade, A. Sanyal, and U. P. Khedker. Validation of GCC optimizers through trace generation. *Softw., Pract. Exper.*, 39(6):611–639, 2009.

[81] M. Kandemir, S. W. Son, and G. Chen. An evaluation of code and data optimizations in the context of disk power reduction. In *Proceedings of the 2005 International Symposium on Low Power Electronics and Design*, pages 209–214, 2005.

[82] M. Kandemir, N. Vijaykrishnan, M. J. Irwin, and W. Ye. Influence of compiler optimizations on system power. *IEEE Trans. Very Large Scale Integr. Syst.*, 9:801–804, December 2001.

[83] M. T. Kandemir. Reducing energy consumption of multiprocessor soc architectures by exploiting memory bank locality. *ACM Trans. Des. Autom. Electron. Syst.*, 11(2):410–441, 2006.

[84] D. M. Kaplan. Some completeness results in the mathematical theory of computation. *J. ACM*, 15(1):124–134, 1968.

[85] M. Karakoy. Optimizing array-intensive applications for on-chip multiprocessors. *IEEE Trans. Parallel Distrib. Syst.*, 16(5):396–411, 2005.

[86] C. Karfa, K. Banerjee, D. Sarkar, and C. Mandal. Equivalence checking of array-intensive programs. In *IEEE Computer Society Annual Symposium on VLSI*, pages 156–161, 2011.

[87] C. Karfa, K. Banerjee, D. Sarkar, and C. Mandal. Experimentation with SMT solvers and theorem provers for verification of loop and arithmetic transformations. In *IBM Collaborative Academia Research Exchange*, pages 3:1–3:4, 2013.

[88] C. Karfa, K. Banerjee, D. Sarkar, and C. Mandal. Verification of loop and arithmetic transformations of array-intensive behaviours. *IEEE Trans. on CAD of ICS*, 32(11):1787–1800, 2013.

[89] C. Karfa, C. Mandal, and D. Sarkar. Verification of register transfer level low power transformations. In *IEEE Computer Society Annual Symposium on VLSI*, pages 313–314, 2011.

[90] C. Karfa, C. Mandal, and D. Sarkar. Formal verification of code motion techniques using data-flow-driven equivalence checking. *ACM Trans. Design Autom. Electr. Syst.*, 17(3):30:1–30:37, 2012.

[91] C. Karfa, C. Mandal, D. Sarkar, S. R. Pentakota, and C. Reade. A formal verification method of scheduling in high-level synthesis. In *International Symposium on Quality Electronic Design*, pages 71–78, 2006.

[92] C. Karfa, C. A. Mandal, D. Sarkar, S. R. Pentakota, and C. Reade. Verification of scheduling in high-level synthesis. In *IEEE Computer Society Annual Symposium on VLSI*, pages 141–146, 2006.

[93] C. Karfa, C. A. Mandal, D. Sarkar, and C. Reade. Register sharing verification during data-path synthesis. In *International Conference on Computing: Theory and Applications*, pages 135–140, 2007.

[94] C. Karfa, D. Sarkar, and C. Mandal. Verification of datapath and controller generation phase in high-level synthesis of digital circuits. *IEEE Trans. on CAD of ICS*, 29(3):479–492, 2010.

[95] C. Karfa, D. Sarkar, C. Mandal, and P. Kumar. An equivalence-checking method for scheduling verification in high-level synthesis. *IEEE Trans. on CAD of ICS*, 27:556–569, 2008.

[96] C. Karfa, D. Sarkar, and C. A. Mandal. Verification of KPN level transformations. In *VLSI Design*, pages 338–343, 2013.

[97] C. Karfa, D. Sarkar, C. A. Mandal, and C. Reade. Hand-in-hand verification of high-level synthesis. In *ACM Great Lakes Symposium on VLSI*, pages 429–434, 2007.

[98] Y. Kim, S. Kopuri, and N. Mansouri. Automated formal verification of scheduling process using finite state machines with datapath (FSMD). In *International Symposium on Quality Electronic Design*, pages 110–115, 2004.

[99] Y. Kim, S. Kopuri, and N. Mansouri. Automated formal verification of scheduling process using finite state machines with datapath (FSMD). In *Proceedings of the 5th International Symposium on Quality Electronic Design*, pages 110–115, 2004.

[100] Y. Kim and N. Mansouri. Automated formal verification of scheduling with speculative code motions. In *ACM Great Lakes Symposium on VLSI*, pages 95–100, 2008.

[101] J. Knoop, O. Ruthing, and B. Steffen. Lazy code motion. In *Programming Language Design and Implementation*, pages 224–234, 1992.

[102] V. Koutavas and M. Wand. Small bisimulations for reasoning about higher-order imperative programs. In *Principles of Programming Languages*, pages 141–152, 2006.

[103] S. Kundu, S. Lerner, and R. Gupta. Validating high-level synthesis. In *Computer Aided Verification*, pages 459–472, 2008.

[104] S. Kundu, S. Lerner, and R. Gupta. Translation validation of high-level synthesis. *IEEE Trans. on CAD of ICS*, 29(4):566–579, 2010.

[105] S. Kundu, Z. Tatlock, and S. Lerner. Proving optimizations correct using parameterized program equivalence. In *Programming Language Design and Implementation*, pages 327–337, 2009.

[106] G. Lakshminarayana, K. Khouri, and N. Jha. Wavesched: A novel scheduling technique for control-flow intensive behavioural descriptions. In *International Conference on Computer-Aided Design*, pages 244–250, 1997.

[107] G. Lakshminarayana, A. Raghunathan, and N. Jha. Incorporating speculative execution into scheduling of control-flow-intensive design. *IEEE Trans. on CAD of ICS*, 19(3):308–324, March 2000.

[108] M. S. Lam and R. P. Wilson. Limits of control flow on parallelism. In *International Symposium on Computer architecture*, pages 46–57, 1992.

[109] B. Landwehr and P. Marwedel. A new optimization technique for improving resource exploitation and critical path minimization. In *International Symposium on Systems Synthesis*, pages 65–72, 1997.

[110] C.-H. Lee, C.-H. Shih, J.-D. Huang, and J.-Y. Jou. Equivalence checking of scheduling with speculative code transformations in high-level synthesis. In *Asia and South Pacific Design Automation Conference*, pages 497–502, 2011.

[111] J.-H. Lee, Y.-C. Hsu, and Y.-L. Lin. A new integer linear programming formulation for the scheduling problem in data path synthesis. In *Procs. of the International Conference on Computer-Aided Design*, pages 20–23, 1989.

[112] X. Leroy, et al. The CompCert C compiler. http://compcert.inria.fr/compcert-C.html.

[113] F. Li and M. T. Kandemir. Locality-conscious workload assignment for array-based computations in MPSOC architectures. In *Design Automation Conference*, pages 95–100, 2005.

[114] Q. Li, L. Shi, J. Li, C. J. Xue, and Y. He. Code motion for migration minimization in STT-RAM based hybrid cache. In *IEEE Computer Society Annual Symposium on VLSI*, pages 410–415, 2012.

[115] T. Li, Y. Guo, W. Liu, and M. Tang. Translation validation of scheduling in high level synthesis. In *ACM Great Lakes Symposium on VLSI*, pages 101–106, 2013.

[116] D. Liu, S. Yin, L. Liu, and S. Wei. Polyhedral model based mapping optimization of loop nests for cgras. In *Design Automation Conference*, pages 19:1–19:8, 2013.

[117] C. A. Mandal and R. M. Zimmer. A genetic algorithm for the synthesis of structured data paths. In *VLSI Design*, pages 206–211, 2000.

[118] Z. Manna. *Mathematical Theory of Computation*. McGraw-Hill Kogakusha, 1974.

[119] T. Matsumoto, K. Seto, and M. Fujita. Formal equivalence checking for loop optimization in C programs without unrolling. In *Advances in Computer Science and Technology*, pages 43–48, 2007.

[120] J. McCarthy. Towards a mathematical science of computation. In *International Federation for Information Processing (IFIP) Congress*, pages 21–28, 1962.

[121] V. Menon, K. Pingali, and N. Mateev. Fractal symbolic analysis. *ACM Trans. Program. Lang. Syst.*, 25(6):776–813, 2003.

[122] R. Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer, 1980.

[123] S.-M. Moon and K. Ebcioğlu. An efficient resource-constrained global scheduling technique for superscalar and VLIW processors. In *Proceedings of the 25th annual international symposium on Microarchitecture*, pages 55–71, 1992.

[124] S. S. Muchnick. *Advanced compiler design and implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997.

[125] K. S. Namjoshi, G. Tagliabue, and L. D. Zuck. A witnessing compiler: A proof of concept. In *Runtime Verification*, pages 340–345, 2013.

[126] G. C. Necula. Translation validation for an optimizing compiler. In *Programming Language Design and Implementation*, pages 83–94, 2000.

[127] A. Nicolau and S. Novack. Trailblazing: A hierarchical approach to percolation scheduling. In *International Conference on Parallel Processing*, volume 2, pages 120–124, Aug. 1993.

[128] M. Palkovic, F. Catthoor, and H. Corporaal. Trade-offs in loop transformations. *ACM Trans. Design Autom. Electr. Syst.*, 14(2):22:1–22:30, 2009.

[129] P. R. Panda, F. Catthoor, N. D. Dutt, K. Danckaert, E. Brockmeyer, C. Kulkarni, A. Vandercappelle, and P. G. Kjeldsberg. Data and memory optimization techniques for embedded systems. *ACM Trans. Des. Autom. Electron. Syst.*, 6:149–206, April 2001.

[130] A. Pnueli, M. Siegel, and E. Singerman. Translation validation. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 151–166, 1998.

[131] M. Potkonjak, S. Dey, Z. Iqbal, and A. C. Parker. High performance embedded system optimization using algebraic and generalized retiming techniques. In *International Conference on Computer Design*, pages 498–504, 1993.

[132] S. Prema, R. Jehadeesan, B. Panigrahi, and S. Satya Murty. Dependency analysis and loop transformation characteristics of auto-parallelizers. In *Parallel Computing Technologies*, pages 1–6, 2015.

[133] D. Qi, A. Roychoudhury, Z. Liang, and K. Vaswani. DARWIN: an approach to debugging evolving programs. *ACM Trans. Softw. Eng. Methodol.*, 21(3):19:1–19:29, 2012.

[134] M. Qiu, E. H.-M. Sha, M. Liu, M. Lin, S. Hua, and L. T. Yang. Energy minimization with loop fusion and multi-functional-unit scheduling for multidimensional DSP. *J. Parallel Distrib. Comput.*, 68(4):443–455, 2008.

[135] M. Rahmouni and A. A. Jerraya. Formulation and evaluation of scheduling techniques for control flow graphs. In *European Design Automation Conference*, pages 386–391, 1995.

[136] M. Rim, Y. Fann, and R. Jain. Global scheduling with code-motions for high-level synthesis applications. *IEEE Trans. VLSI Syst.*, 3(3):379–392, 1995.

[137] G. Roy. Techniques and Algorithms for the Design and Development of a Virtual Laboratory to Support Logic Design and Computer Organization. Master's thesis, IIT Kharagpur, India, 2014.

[138] O. Ruthing, J. Knoop, and B. Steffen. Sparse code motion. In *Principles of Programming Languages*, pages 170–183, 2000.

[139] H. Samsom, F. Franssen, F. Catthoor, and H. De Man. System level verification of video and image processing specifications. In *International Symposium on Systems Synthesis*, pages 144–149, 1995.

[140] D. Sangiorgi. On the origins of bisimulation and coinduction. *ACM Trans. Program. Lang. Syst.*, 31(4):15:1–15:41, 2009.

[141] D. Sarkar and S. De Sarkar. A theorem prover for verifying iterative programs over integers. *IEEE Trans. Software. Engg.*, 15(12):1550–1566, 1989.

[142] P. Sarkar and S. Maitra. Construction of nonlinear Boolean functions with important cryptographic properties. In *International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, pages 485–506, 2000.

[143] K. K. Sharma, K. Banerjee, and C. Mandal. A scheme for automated evaluation of programming assignments using FSMD based equivalence checking. In *IBM Collaborative Academia Research Exchange*, pages 10:1–10:4, 2014.

[144] K. C. Shashidhar. *Efficient Automatic Verification of Loop and Data-flow Transformations by Functional Equivalence Checking*. PhD thesis, Katholieke Universiteit Leuven, 2008.

[145] K. C. Shashidhar, M. Bruynooghe, F. Catthoor, and G. Janssens. Geometric model checking: An automatic verification technique for loop and data reuse transformations. *Electronic Notes in Theoretical Computer Science*, 65(2):71–86, 2002.

[146] K. C. Shashidhar, M. Bruynooghe, F. Catthoor, and G. Janssens. Functional equivalence checking for verification of algebraic transformations on array-intensive source code. In *Design, Automation and Test in Europe*, pages 1310–1315, 2005.

[147] K. C. Shashidhar, M. Bruynooghe, F. Catthoor, and G. Janssens. Verification of source code transformations by program equivalence checking. In *Compiler Construction*, pages 221–236, 2005.

[148] R. E. Tarjan. Depth-first search and linear graph algorithms. *SIAM J. Comput.*, 1(2):146–160, 1972.

[149] S. Tripakis and S. Yovine. Analysis of timed systems using time-abstracting bisimulations. *Formal Methods in System Design*, 18(1):25–68, 2001.

[150] J.-B. Tristan and X. Leroy. Verified validation of lazy code motion. In *Programming Language Design and Implementation*, pages 316–326, 2009.

[151] P. Vanbroekhoven, G. Janssens, M. Bruynooghe, and F. Catthoor. Transformation to dynamic single assignment using a simple data flow analysis. In *Asian symposium on Programming Languages and Systems*, pages 330–346, 2005.

[152] S. Verdoolaege. ISL: An Integer Set Library for the Polyhedral Model. In *International Congress on Mathematical Software*, pages 299–302, 2010.

[153] S. Verdoolaege, G. Janssens, and M. Bruynooghe. Equivalence checking of static affine programs using widening to handle recurrences. In *Computer Aided Verification*, pages 599–613, 2009.

[154] S. Verdoolaege, G. Janssens, and M. Bruynooghe. Equivalence checking of static affine programs using widening to handle recurrences. *ACM Trans. Program. Lang. Syst.*, 34(3):11:1–11:35, 2012.

[155] T. Šimunić, L. Benini, G. De Micheli, and M. Hans. Source code optimization and profiling of energy consumption in embedded systems. In *International Symposium on Systems Synthesis*, pages 193–198, 2000.

[156] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. B. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. P. Puschner, J. Staschulat, and P. Stenström. The worst-case execution-time problem - overview of methods and survey of tools. *ACM Trans. Embedded Comput. Syst.*, 7(3):36:1–36:53.

[157] R. Wimmer, M. Herbstritt, H. Hermanns, K. Strampp, and B. Becker. Sigref- A symbolic bisimulation tool box. In *Automated Technology for Verification and Analysis*, pages 477–492, 2006.

[158] L. Xue, O. Ozturk, and M. Kandemir. A memory-conscious code parallelization scheme. In *Design Automation Conference*, pages 230–233, 2007.

[159] Y.-P. You and S.-H. Wang. Energy-aware code motion for gpu shader processors. *ACM Trans. Embed. Comput. Syst.*, 13(3):49:1–49:24, 2013.

[160] H. Yviquel, A. Sanchez, P. Jääskeläinen, J. Takala, M. Raulet, and E. Casseau. Embedded multi-core systems dedicated to dynamic dataflow programs. *Signal Processing Systems*, 80(1):121–136, 2015.

[161] S. Zamanzadeh, M. Najibi, and H. Pedram. Pre-synthesis optimization for asynchronous circuits using compiler techniques. In *Advances in Computer Science and Engineering*, volume 6 of *Communications in Computer and Information Science*, pages 951–954. Springer Berlin Heidelberg, 2009.

[162] C. Zhang and F. Kurdahi. Reducing off-chip memory access via stream-conscious tiling on multimedia applications. *Int. J. Parallel Program.*, 35(1):63–98, 2007.

[163] J. Zhao, S. Nagarakatte, M. M. K. Martin, and S. Zdancewic. Formal verification of SSA-based optimizations for LLVM. In *Programming Language Design and Implementation*, pages 175–186, 2013.

[164] Y. Zhu, G. Magklis, M. L. Scott, C. Ding, and D. H. Albonesi. The energy impact of aggressive loop fusion. In *Proceedings of the 13th International*

*Conference on Parallel Architectures and Compilation Techniques*, pages 153–164, 2004.

[165] J. Zory and F. Coelho. Using algebraic transformations to optimize expression evaluation in scientific codes. In *International Conference on Parallel Architectures and Compilation Techniques*, pages 376–384, 1998.

[166] L. Zuck, A. Pnueli, Y. Fang, and B. Goldberg. Voc: A translation validator for optimizing compilers. *Journal of Universal Computer Science*, 9(3):223–247, 2003.

[167] L. D. Zuck, A. Pnueli, B. Goldberg, C. W. Barrett, Y. Fang, and Y. Hu. Translation and run-time validation of loop transformations. *Formal Methods in System Design*, 27(3):335–360, 2005.

# Publications:

## Journal/conference papers:

1. **Kunal Banerjee**, Dipankar Sarkar, Chittaranjan Mandal; “Deriving Bisimulation Relations from Path Extension Based Equivalence Checkers;” *IEEE Transformations on Software Engineering (TSE)*, (accepted).

2. **Kunal Banerjee**, Dipankar Sarkar, Chittaranjan Mandal; “Deriving Bisimulation Relations from Value Propagation Based Equivalence Checkers of Programs;” *Formal Aspects of Computing (FAOC)*, vol. 29, no. 2, 2017, pages: 365–379.

3. **Kunal Banerjee**, Dipankar Sarkar, Chittaranjan Mandal; “Extending the FSMD Framework for Validating Code Motions of Array-Handling Programs;” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, vol. 33, no. 12, 2014, pages: 2015–2019.

4. **Kunal Banerjee**, Chandan Karfa, Dipankar Sarkar, Chittaranjan Mandal; “Verification of Code Motion Techniques using Value Propagation;” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, vol. 33, no. 8, 2014, pages: 1180–1193.

5. **Kunal Banerjee**, Chittaranjan Mandal, Dipankar Sarkar; “An Equivalence Checking Framework for Array-Intensive Programs;” *Automated Technology for Verification and Analysis (ATVA)*, Pune, India, 2017, (accepted).

6. **Kunal Banerjee**, Chittaranjan Mandal, Dipankar Sarkar; “Translation Validation of Loop and Arithmetic Transformations in the Presence of Recurrences;” *ACM SIGPLAN/SIGBED Conference on Languages, Compilers, Tools and Theory for Embedded Systems (LCTES)*, Santa Barbara, United States, 2016, pages: 31–40.

7. **Kunal Banerjee**, Chittaranjan Mandal, Dipankar Sarkar; “A Translation Validation Framework for Symbolic Value Propagation Based Equivalence Checking of FSMDAs;” *Source Code Analysis and Manipulation (SCAM)*, Bremen, Germany, 2015, pages: 247–252.

8. **Kunal Banerjee**, Chittaranjan Mandal, Dipankar Sarkar; "Translation Validation of Transformations of Embedded System Specifications using Equivalence Checking;" *IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, Montpellier, France, 2015, pages: 183–186, **(received Best PhD Forum Paper Award)**.

9. **Kunal Banerjee**, Chandan Karfa, Dipankar Sarkar, Chittaranjan Mandal; "A Value Propagation Based Equivalence Checking Method for Verification of Code Motion Techniques;" *International Symposium on Electronic System Design (ISED)*, Kolkata, India, 2012, pages: 67–71.

## Publications in research fora:

Note that the following dissemination arising out of this work were not published as part of the conference/workshop proceedings; these venues rather aimed to provide a platform for young researchers to discuss their works with experts in their respective fields; all of these work, however, went through standard peer review process before being accepted.

10. **Kunal Banerjee**; "Translation Validation of Transformations of Embedded System Specifications using Equivalence Checking;" *Inter-Research-Institute Student Seminar in Computer Science (IRISS)*, Goa, India, 2015.

11. **Kunal Banerjee**; "An Equivalence Checking Mechanism for Handling Recurrences in Array-Intensive Programs;" *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL): Student Research Competition*, Mumbai, India, 2015.

12. **Kunal Banerjee**, Chittaranjan Mandal, Dipankar Sarkar; "Deriving Bisimulation Relations from Path Extension Based Equivalence Checkers;" *IMPECS-POPL Workshop on Emerging Research and Development Trends in Programming Languages (WEPL)*, Mumbai, India, 2015.