Scheduling Verification in High-Level Synthesis - Implementation of a Normalizer and a Code Motion Verifier

THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF

Master of Technology
in
Computer Science and Engineering and Information Technology

by

Pramod Kumar
03CS3019

Under the guidance of

Prof. C. R. Mandal
and
Prof. Dipankar Sarkar

" YOGA KARMASU
KAUSALAM "

Department of Computer Science and Engineering
Indian Institute of Technology Kharagpur
May 2008

Certificate

This is to certify that the thesis titled "***Scheduling Verification in High-Level Synthesis - Implementation of a Normalizer and a Code Motion Verifier***" submitted by Pramod Kumar, to the Department of Computer Science and Engineering, Indian Institute of Technology, Kharagpur, India, in partial fulfillment for the award of the degree of Master of Technology, is a bonafide record of original research work carried out by him under our supervision and guidance. The thesis fulfills all the requirements as per the regulations of this Institute and, in our opinion, has reached the standard needed for submission. Neither this thesis nor any part of it has been submitted for any degree elsewhere.

Place: I.I.T. Kharagpur	Prof. C. R. Mandal
	Dept. of Computer Science and Engg
Date:	Indian Institute of Technology Kharagpur
	721302, INDIA.

Place: I.I.T. Kharagpur	Prof. Dipankar Sarkar
	Dept. of Computer Science and Engg
Date:	Indian Institute of Technology Kharagpur
	721302, INDIA.

Acknowledgment

This thesis is the result of my research work under the guidance of Prof. C. R. Mandal and Prof. Dipankar Sarkar at the Department of Computer Science and Engineering of the Indian Institute of Technology, Kharagpur. I am deeply thankful to my research advisors for the huge amount of time and effort they spent guiding me through several difficulties on the way. Without the help, encouragement and patient support I received from my guides, this thesis would never have materialized.

In particular, I would like to thank Mr. Chandan Karfa for his encouragement and support throughout this period.

Pramod Kumar
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur
May 2008

Abstract

High level synthesis is the process of generating the register transfer level (RTL) design from the behavioral description. The synthesis process consists of several interdependent phases: Preprocessing, Scheduling, Register Allocation and Binding of variables, Control Path and Data Path generation, and Generation of synthesizable Verilog code (RTL).

A High-level synthesis tool, called Structured Architecture Synthesis tool (SAST), has been developed which support hand-in-hand synthesis and verification. The existing framework is the SAST, and this work is to enhance this tool by incorporating a Normalizer and a Code Motion Verifier.

The complexity of present-day VLSI systems is very high. The specification is given at a high level of abstraction compared to that of the output. In addition several optimization and transformations may be made at each phase to improve the performance of the design. Hence it is important to ensure that after each phase the behavior of the original specification is preserved. Hence is the need of phase-wise verification.

Verification of high level synthesis is a formal method for checking the equivalence between two descriptions of the target system, one before a particular phase and the other after that phase. The descriptions are represented as Finite State Machines with Data paths (FSMD).The basic principle is to show that any computation of one FSMD is covered by a computation on the other.

While finding the equivalent path for a path, it is required to check the equivalence of the respective conditions as well as the data transformations of the paths. Since the condition of execution and the data transformation of a path involve the whole of integer arithmetic, checking equivalence of paths reduces to the validity problem of first order logic which is undecidable; thus, a canonical form does not exist for integer arithmetic. There we use a normalized form for conditional expression and data transformation expression.

It may be possible to transform the input behavior to some equivalent description, by incorporating several high-level code transformation techniques, which results in amore efficient scheduling behavior. Thus the need to enhance the verifier to handle various code motion techniques while verification.

Key Words: High-level Synthesis, Verification, Equivalence Checking, Normalization, FSMD models, Code Motion Techniques, SAST.

Contents

Chapter 1

Introduction

1.1 High Level Synthesis

High level synthesis is a process to translate a behavioral specification into RTL description. It takes a CDFG as input and undergoes some processes and produces output as a data path and a controller, so that the data transfers under the controller exhibiting the specified behavior.

High level synthesis consists of several phases:

- Preprocessing: Translation of the input control data flow graph (CDFG) to an intermediate representation (IR) and calculation of necessary information for scheduling.
- Schedule of the operations and the transfer of variables in minimum number of control steps for a given architectural specification. The scheduler accomplishes functional unit formation.
- Allocation and binding of variables to registers.
- Data path generation from the schedule of operations, bus transfers and the variable mapping to the registers.
- Generation of synthesizable Verilog code (RTL).

A High-level synthesis tool, called Structured Architecture Synthesis tool (SAST), has been developed which support hand-in-hand synthesis and verification.

1.2 Phase Wise Verification:

Formalization of a general verification methodology can be applied to all the phases of High Level Synthesis. It handles the difficulties of each phase. The input and output of every phase of HLS is represented by FSMD's. The phase wise verification methodology is based on the equivalence problem of two FSMD's, the FSMD before the phase and the FSMD after the phase.

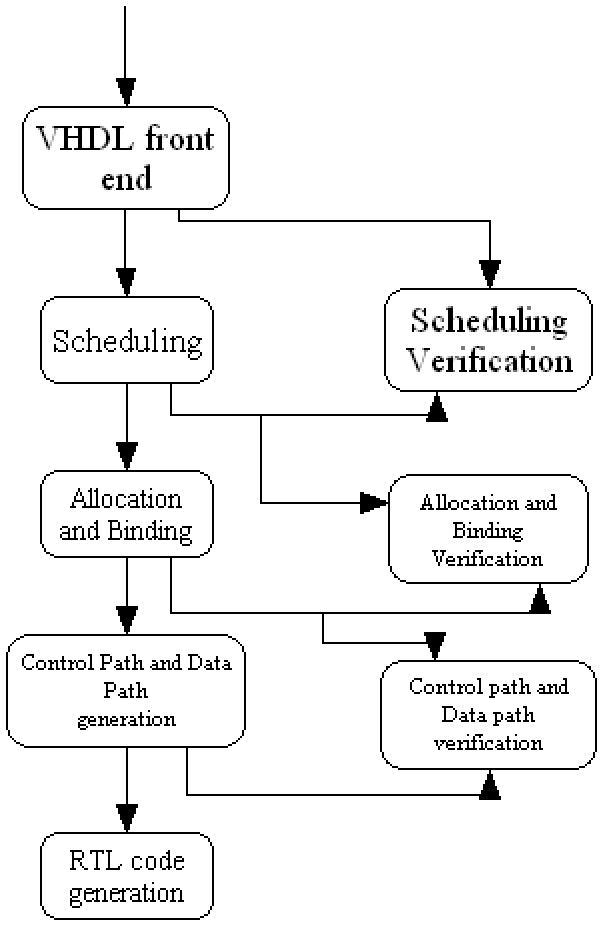

VHDL front end
Scheduling
Scheduling Verification
Allocation and Binding
Allocation and Binding Verification
Control Path and Data Path generation
Control path and Data path verification
RTL code generation

Fig. Phase wise verification.

1.3 Motivation of the Present Work:

- The complexity of present day VLSI systems is very high. The Specification is given at a high level of abstraction compared to the output. In addition different types of optimization are performed in each phase of the High-Level Synthesis. The input behavior to the scheduler is modified in several ways in order to use a minimum number of time steps to schedule the operations. Also, incorporation of several code-motion techniques in the scheduling process leads to moving operations across the basic Blocks (BB) boundaries. Consequently, the results of the scheduling do not have a one to one correspondence with the input.

- Finding equivalence between two FSMDs involves finding equivalence between two paths which in turn involves checking equivalence between two sets of arithmetic expressions. Hence, checking equivalence of two paths reduces to the validity problem of first-order logic which is undecidable; thus, a canonical form does not exist for integer arithmetic. Instead, in this work we adapt a normal form for the arithmetic expressions over integers. The normalization process renders many computationally equivalent formulas syntactically identical as it forces all the formulas to follow a uniform structure.

1.4 Contribution of the Present Work:

- **Implementation of a Normalizer.**
- **Implementation of a code-motion verifier.**
- **Implementation of a test bench (** ***IEEE754*** ***)***

Chapter 2

Overview of FSMDs and their Equivalence

2.1 Finite State machines With Data Paths (FSMDs)

An FSMD is a universal specification model that can represent all hardware designs. The FSMD is defined as an ordered tuple $< Q, q0, I, V, O, f, h >$, where

1) $Q = \{q0, q1, q2, \ldots, qn\}$ is the finite set of control states,
2) $q0 \in Q$ is the reset state,
3) I is the set of primary input signals,
4) V is the set of storage variables, and Σ is the set of all data storage states or simply data states,
5) O is the set of primary output signals,
6) $f : Q \times 2^S \rightarrow Q$ is the state transition function,
7) $h : Q \times 2^S \rightarrow U$ is the update function of the output and the storage variables, where S and U are defined as follows:

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

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

The implicit connective among the members of the set of status expressions, which occurs as the second argument of the function f (or h), is conjunction. Parallel edges between two states capture the disjunction of status expressions. Thus, the next (control and data) state and the output depend not only on the present state and the input signals but also on the conjunction of the status expressions that indicate whether a predicate holds on the data state of the storage and the input variables. The state transition function and the update function are such that the FSMD model remains deterministic. Thus, for any state q, if $f(q, S1)$ and $f(q, S2)$ are different, then the sets $S1$ and $S2$ of status expressions are disjointed. The same property holds for the update function h. It may be noted that we have not introduced any final state in the FSMD model as we assume that a system works in an infinite outer loop.

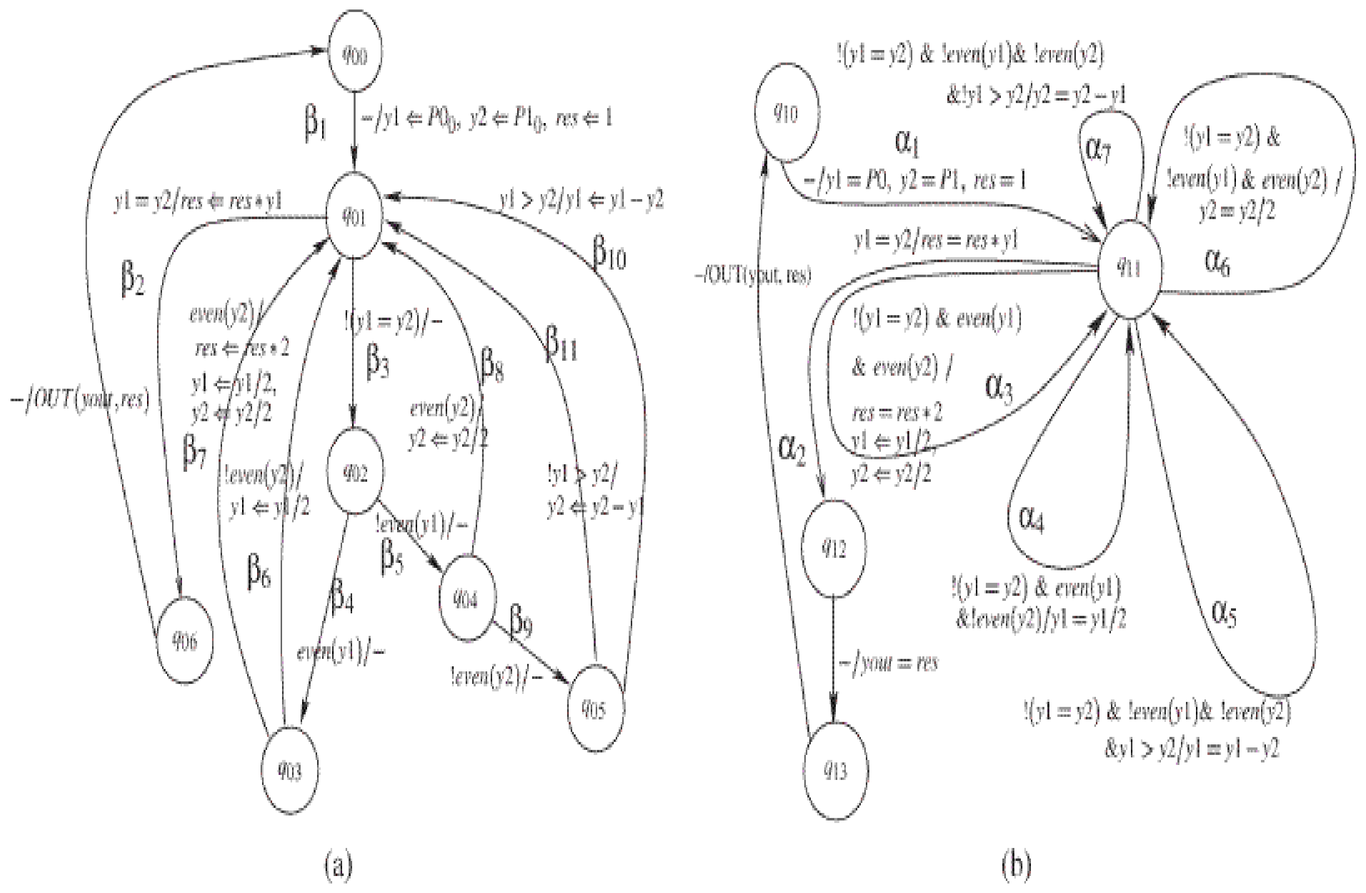

q00
q01
q02
q03
q04
q05
q06
β1
−/y1 ⇐ P0₀, y2 ⇐ P1₀, res ⇐ 1
y1 = y2/res ⇐ res ∗ y1
β2
y1 > y2/y1 ⇐ y1 − y2
β10
even(y2)/ res ⇐ res ∗ 2 y1 ⇐ y1/2, y2 ⇐ y2/2
β7
!(y1 = y2)/−
β3
β8
β11
−/OUT(yout, res)
even(y2)/ y2 ⇐ y2/2
!even(y2)/ y1 ⇐ y1/2
!y1 > y2/ y2 ⇐ y2 − y1
!even(y1)/−
β5
β6
β4
even(y1)/−
β9
!even(y2)/−
(a)
q10
q11
q12
q13
!(y1 = y2) & !even(y1)& !even(y2) &!y1 > y2/y2 = y2 − y1
α1
α7
−/y1 = P0, y2 = P1, res = 1
!(y1 = y2) & !even(y1) & even(y2) / y2 = y2/2
y1 = y2/res = res ∗ y1
α6
−/OUT(yout, res)
!(y1 = y2) & even(y1) & even(y2) / res = res ∗ 2 y1 ⇐ y1/2, y2 ⇐ y2/2
α3
α2
α4
!(y1 = y2) & even(y1) &!even(y2)/y1 = y1/2
α5
−/yout = res
!(y1 = y2) & !even(y1)& !even(y2) &y1 > y2/y1 = y1 − y2
(b)

Fig. (a) *M0:* FSMD of GCD before scheduling.
(b) *M1*: FSMD of GCD after scheduling

Example 1:
The FSMD model *M*0 for the behavioral specification of GCD example is depicted in Fig. (a). Specifically,
1) *M*0 = *<Q, q*0*, I, V, O, f, h>*.
2) *Q* = *{q*00*, q*01*, q*02*, q*03*, q*04*, q*05*, q*06*}*, *q*0 = *q*00, *V* = *{*res*, y*1*, y*2*}*, *I* = *{P*0*, P*1*}*, and *O* = *{*yout*}*.
3) $U = \{y1 \Leftarrow P0,\ y2 \Leftarrow P1,$ res $\Leftarrow 1$ res $\Leftarrow$ res $\ast 2, = y1 \Leftarrow y1/2,\ y2 \Leftarrow y2/2,\ y1 \Leftarrow y1 - y2,\ y2 \Leftarrow y2 - y1,$ res $\Leftarrow$ res $\ast y1\}$.
4) *S* = *{*even(*y*1)*,* even(*y*2)*, y*1 == *y*2*, y*1 > *y*2*}*, where even(*y*) is the abbreviation of "*y* mod 2 = 0."

5) f and h are as defined in the transition graph shown in Fig. (a).
6) Some typical values of f and h are as follows.

a) $f(q00, \{\text{true}\}) = q01$.
b) $f(q05, \{y1 > y2\}) = q01$.
c) $h(q05, \{y1 > y2\}) = \{y1 \Leftarrow y1 - y2\}$.
d) $h(q04, \{\text{even}(y2)\}) = \{y2 \Leftarrow y2/2\}$.

2.2 Paths and Transformations along a Path

A (finite) path α from qi to qj , where $qi, qj \in Q$, is a finite transition sequence of states of the form $<qi = q1 \xrightarrow{}_{c1} q2 \xrightarrow{}_{c2} . . . \xrightarrow{}_{cn-1} qn = qj>$ such that $\forall l, 1 \leq l \leq n - 1, \exists cl \in 2^S$ such that $f(ql, cl) = ql+1$, and $qk, 1 \leq k \leq n - 1$, are all distinct. The state qn may be identical to any $qk, 1 \leq k \leq n - 1$. The condition of execution $R\alpha$ of the path $\alpha = <q1 \xrightarrow{}_{c1} q2 \xrightarrow{}_{c2} . . . \xrightarrow{}_{cn-1} qn>$ is a logical expression over $I \cup V$ such that $R\alpha$ is satisfied by the (initial) data state at $q1$ if the path α is traversed. Thus, $R\alpha$ is the weakest precondition of the path α. Often, for brevity, the aforementioned path α is represented as $[q1 \Rightarrow qn]$.

We assume that inputs and outputs occur through named ports. The *ith* input from port P is a value symbolically represented as Pi. Thus, if some variable v stores an input from port P (for the *ith* time along a path), it is equivalent to the assignment $v \Leftarrow Pi$. In essence, Pi's comprise the input variable set I and each input variable in I is read only once in a computation (path).

The simple data transformation $s\alpha$ of a path α over V is an ordered tuple $<ej>$ of algebraic expressions over $I \cup V$ such that the expression ej represents the value of the variable vj after execution of the path in terms of the initial data state (i.e., the values of the variables at the initial control state) of the path. Taking into account the outputs that may occur in a path, the data transformation rα of a path α over V is the tuple <sα, Oα>, where the output list $O\alpha = [OUT(Pi1, e1), OUT(Pi2, e2), . . .]$. More specifically, for every expression e output to port P along the path α, there is a member $OUT(P, e)$ in the list appearing in the order in which the outputs occur in α.

2.3 Computation of *Rα* and *rα*

Computation of the condition of execution $R\alpha$ can be obtained by backward substitution or by forward substitution. The former is more easily perceivable and is based on the following rule: If a predicate $c(y)$ is true after the assignment $y \leftarrow g(y)$, then the predicate $c(g(y))$ must have been true before the assignment. The transformation $s\alpha$ is found indirectly using the same principle. The forward-substitution method of finding $R\alpha$ is based on symbolic execution. The ordered pairs at various points in Fig. represent the values of $(R\alpha, s\alpha)$ at that point.

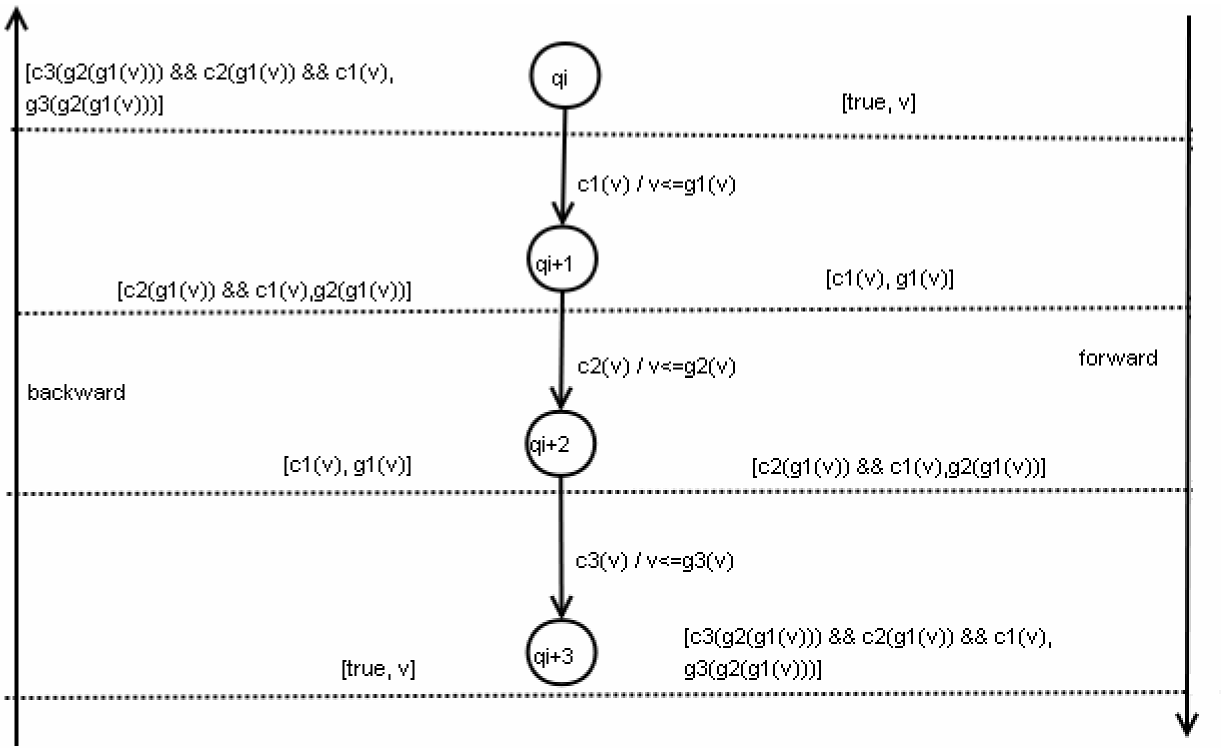


Fig. Typical path, its condition of execution, and its simple data transformation.

2.4 Characterization of Paths

The characteristic formula $\tau\alpha(v', v'f, O)$ of the path α is $R\alpha(v') \wedge (v'f = s\alpha(v')) \wedge (O = \alpha(v'))$, where $s\alpha$ is the data transformation, $O\alpha$ is the output list in the path α, v' represents a vector of variables of $I\ U\ V$ and $v'f$ represents a vector of variables of V . The formula captures the following: If the condition of execution $R\alpha$ of the path α is satisfied by the (initial) vector v' at the beginning of the path, then the path is executed, and after execution, the final vector $v'f$ of variable values becomes $s\alpha(v')$, and the output $O\alpha(v')$ is produced.

Let $\tau\alpha(v', v'f, O) : R\alpha(v') \wedge (v'f = s\alpha(v')) \wedge (O=O\alpha(v'))$ be the characteristic formula of the path α and $\tau\beta(v', v'f, O) : R\beta(v') \wedge (v'f = s\beta(v')) \wedge (O = O\beta(v'))$ be the characteristic formula of the path β. The characteristic formula for the concatenated path $\alpha\beta$ is $\tau\alpha\beta(v', v'f, O) = \exists v'\alpha\ \exists O1\ \exists O2\ (\tau\alpha(v', v'\alpha, O1) \wedge \tau\beta(v'\alpha, v'f, O2)) = R\alpha(v') \wedge R\beta(s\alpha(v')) \wedge (v'f = s\beta(s\alpha(v'))) \wedge (O = O\alpha(v')O\beta(s\alpha(v')))$. O is the concatenated output list of $O\alpha(v')$ and $O\beta(s\alpha(v'))$. It is necessary to increment the input indices on each port in the formulas for β to start after the last index of the corresponding port in α.

2.5 Computations and Path Covers of an FSMD

A computation of an FSMD is a finite walk from the reset state $q0$ back to itself without having any intermediary occurrence of $q0$. Such a computational semantics of an FSMD is based on the assumption that a revisit of the reset state means the beginning of a new computation and that each computation terminates. A computation μ of an FSMD M may be characterized as $\tau\mu(v'i, v'f, O) : R\mu(v'i) \wedge (v'f = s\mu(v'i)) \wedge (O = O\mu(v'i))$, where $v'i$ is the vector of the initial input with which the computation is started, $R\mu$ is a satisfiable condition over the domain of I, $s\mu$ is a function over this domain to the co domain of values over V and $O\mu$ is the concatenation of the output lists resulting from output operations along μ. The ordered pair $<s\mu, O\mu>$ is denoted as $r\mu$.

Definition 1: Two computations $\mu1$ and $\mu2$ having the characteristic formula $\tau\mu1$ and $\tau\mu2$, respectively, are said to be equivalent, denoted as $\mu1 \equiv \mu2$, if $R\mu1 = R\mu2$ and $r\mu1 = r\mu2$.

The computational equivalence of two paths $p1$ and $p2$ can be defined in a similar manner and is denoted as $p1 \equiv p2$. Equivalence checking of paths, therefore, consists in establishing the computational equivalence of the respective condition of execution and the respective data transformation.

Any computation μ of an FSMD M can be looked upon as a computation along some concatenated path $[\alpha1\alpha2\alpha3, . . . , \alpha k]$ of M such that the path $\alpha1$ emanates from and the path αk terminates in the reset state $q0$ of M for $1 \leq i \leq k$, αi terminates in the initial state of the path αi+1, and αi's may not all be distinct. The characteristic formula $\tau\mu$ of μ can accordingly be defined in terms of characteristic formula of the concatenated paths corresponding to μ. Hence, we have the following definition.

Definition 2—Path Cover of an FSMD: A finite set of paths $P = \{p0, p1, p2, . . . , pk\}$ is said to be a path cover of an FSMD M if any computation μ of M can be looked upon as a concatenation of paths from P.

2.6 Equivalence of FSMDs

Let the behavior given as input to the scheduler be represented by the FSMD $M0 = <Q0, q00, I, V0, O, f0, h0>$ and the scheduled behavior be represented by the FSMD $M1 = <Q1, q10, I, V1, O, f1, h1>$. Our main goal is to verify whether $M0$ behaves exactly as $M1$. This means that for all possible input sequences, $M0$ and $M1$ produce the same sequences of output values and eventually, when the respective reset states are revisited, they are visited with the same storage element values. In other words, for every computation from the reset state back to itself of one FSMD, there exists an equivalent computation from the reset state back to itself in the other FSMD and vice versa.

Definition 3: An FSMD $M0$ is said to be contained in an FSMD $M1$, symbolically $M0 \sqsubseteq M1$, if, for any computation $\mu0$ of $M0$, there exists a computation $\mu1$ of $M1$ such that $\mu0 \equiv \mu1$.

Definition 4: Two FSMDs $M0$ and $M1$ are said to be computationally equivalent if $M0 \sqsubseteq M1$ and $M1 \sqsubseteq M0$.

Theorem 1: An FSMD $M0$ is contained in another FSMD $M1$ ($M0 \sqsubseteq M1$) if there exists a finite cover $P0 = \{p00, p01, \ldots, p0l\}$ of $M0$ for which there exists a set $P1 = \{p10, p11, \ldots, p1l\}$ of paths of $M1$ such that $p0i \equiv p1i$, $0 \leq i \leq l$.

Proof: $M0 \sqsubseteq M1$ if, for any computation $\mu 0$ of $M0$, there exists a computation $\mu 1$ of $M1$ such that $\mu 0$ and $\mu 1$ are computationally equivalent [by Definition 3].

Now, let there exist a finite cover $P0 = \{p00, p01, \ldots, p0l\}$ of $M0$. Corresponding to $P0$, let a set $P1 = \{p10, p11, \ldots, p1l\}$ of paths of $M1$ exist such that $p0i \equiv p1i$, $0 \leq i \leq l$.

Since $P0$ covers $M0$, any computation $\mu 0$ of $M0$ can be looked upon as a concatenated path $[p0i1 p0i2 \cdots p0in]$ from $P0$ starting from the reset state $q00$ and ending again at this reset state of $M0$. From the above hypothesis, it follows that there exists a sequence $\Pi 1$ of paths $[p1j1 p1j2 \ldots p1jn]$ of $P1$, where $p0ik \equiv p1jk$, $1 \leq k \leq n$. Therefore, in order that $\Pi 1$ represents a computation of $M1$, it is required to prove that $\Pi 1$ is a concatenated path of $M1$ from its reset state $q10$ back to itself. The following definition is in order.

Definition 5—Corresponding States: Let $M0 = <Q0, q00, I, V0, O, f0, h0>$ and $M1 = <Q1, q10, I, V1, O, f1, h1>$ be the two FSMDs having identical input and output sets, I and O, respectively, and $q0i, q0k \in Q0$ and $q1j, q1l \in Q1$.

1) The respective reset states $q00$ and $q10$ are corresponding states.

2) If $q0i \in Q0$ and $q1j \in Q1$ are corresponding states and there exist $q0k \in Q0$ and $q1l \in Q1$ such that, for some path α from $q0i$ to $q0k$ in $M0$, there exists a path β from $q1j$ to $q1l$ in $M1$ such that $\alpha \equiv \beta$, then $q0k$ and $q1l$ are corresponding states.

Now, let $p0i1 : [q00 \Rightarrow q0f1]$. Since $p1j1 \equiv p0i1$, from the above definition of corresponding states, $p1j1$ must be of the form $[q10 \Rightarrow q1f1]$, where $<q00, q10>$ and $<q0f1, q1f1>$ are corresponding states. Thus, by repetitive application of the above argument, it follows that if $p0i1 : [q00 \Rightarrow q0f1]$, $p0i2 : [q0f1 \Rightarrow q0f2]$, $\ldots$, $p0in$: $[q0fn{-}1 \Rightarrow q0fn = q00]$, then $p1i1 : [q10 \Rightarrow q1f1]$, $p1i2 : [q1f1 \Rightarrow q1f2]$, $\ldots$, $p1in$: $[q1fn{-}1 \Rightarrow q1fn = q10]$, where $<q0fm, q1fm>$, $1 \leq m \leq n$, are pairs of corresponding states. Hence, $\Pi 1$ is a concatenated path representing a computation $\mu 1$ of $M1$, where $\mu 1 \equiv \mu 0$.

2.7 Equivalence of Paths

Theorem 1 reduces the equivalence problem of two FSMDs into the problem of determining whether a path of one FSMD is equivalent to some path of the other FSMD. A computation $\mu 0$ of FSMD $M0$ can be compared with a computation $\mu 1$ of $M1$ as long as both $M0$ and $M1$ have the same input variable set I because $R\mu 0$, $R\mu 1$, $r\mu 0$, and $r\mu 1$ are all defined over I. In contrast, a comparison of a path $p0$ of $M0$ with a path $p1$ of $M1$, however, is not so straightforward because, in general, $M0$, $M1$ may involve different storage variable sets $V0$ and $V1$, respectively. This may happen due to various code-motion techniques applied during scheduling. Since paths can start from any cut points of the FSMD, their conditions of execution and the data transformations will be in terms of the inputs and the storage variables. Thus, the path $p0$ may involve variables from the set $V0$ and the path $p1$ may involve variables from $V1 \neq V0$. To handle such situations, the condition $R\alpha$, the data transformation $s\alpha$ and the output list $O\alpha$ of any path α should also be restricted over the variables $V0 \cap V1$ and the inputs I.

Any expression (arithmetic or status) is *defined* over the variable set $V0 \cap V1$ if all the variables it involves belong to $V0 \cap V1$. A tuple of expressions over $V0$ or $V1$, restricted to $V0 \cap V1$, is its projection over $V0 \cap V1$ in which all the component expressions are defined over $V0 \cap V1$. The restrictions of $R\alpha$ and $r\alpha$ of a path α follow from the restrictions of their constituent expressions as defined above. The restrictions of the condition of execution and the data transformation of a path α on the variable set $V0 \cap V1$ are denoted as $R\alpha|V0 \cap V1$ and $r\alpha|V0 \cap V1$, respectively. For example, let $V0 = \{v0, v1, v2\}$ and $V1 = \{v1, v2, v3\}$. Therefore, $V0 \cap V1$ is $\{v1, v2\}$. Let the condition of execution of a path in $M1$ be ($v1 - v2 > 0 \wedge v1 \leq v2+v3$); under restriction to $V1$, the condition becomes undefined as $v3$ occurs in the conditional expression. Let the data transformation of a path in $M1$ be $<<v1 - v2, v2 + 2, v3 - v1>, ->$, where the order of the variables is $v1 \prec v2 \prec v3$. Under restriction to $\{v1, v2\}$, the transformation becomes $<<v1 - v2, v2 + 2>, ->$. Thus, the transformation of this path under restriction to $V0 \cap V1$ is defined even if the final value of the variable $v3$ is not restricted to $V0 \cap V1$ because the final values of $v0$ and $v3$ are not considered during checking the equality of the data transformations of paths of $M0$ and $M1$. Consider another path in $M1$ whose data transformation is $<<v1 - 1, v2 + v3, v2 + 1>, ->$. This transformation becomes undefined when restricted to $\{v1, v2\}$ as $v3$ occurs in the expression value of $v2 \in V0 \cap V1$. Hence, we have the following definition.

Definition 6: A path α of $M0 = <Q0, q00, I, V0, O, f0, h0>$ and a path β of $M1 = <Q1, q10, I, V1, O, f1, h1>$ are said to be equivalent if $R\alpha$, $r\alpha$, $R\beta$ and $r\beta$ are defined over $V0 \cap V1$ and $R\alpha|V0 \cap V1 = R\beta|V0 \cap V1$ and $r\alpha|V0 \cap V1 = r\beta|V0 \cap V1$. The $R\alpha|V0 \cap V1$, $R\beta|V0 \cap V1$, $r\alpha|V0 \cap V1$, and $r\beta|V0 \cap V1$ become undefined even if the scheduler transformations are correct when some of the variables (in $V0 \cap V1$) are eliminated or some of the variables (in $V1 \cap V0$) are introduced (defined) prior to the start state of the path under consideration. In these cases, our algorithm reports false negative.

2.8 VERIFICATION METHOD

Theorem 1 suggests a verification method for checking equivalence of two FSMDs which consists of the following steps.

1) Construct the set $P0$ of paths of $M0$ so that $P0$ covers $M0$. Let $P0 = \{p00, p01, \ldots, p0k\}$.

2) Show that $\forall p0i \in P0$, there exists a path $p1j$ of $M1$ such that $p0i \equiv p1j$.

3) Repeat steps 1 and 2 with *M*0 and *M*1 interchanged.

Because of loops, it is difficult to find a path cover of the whole computation comprising only finite paths. Therefore, any computation is split into paths by putting *cutpoints* at various places in the FSMD so that each loop is cut in at least one cutpoint. The set of all paths from a cutpoint to another cutpoint without having any intermediary cutpoint is a path cover of the FSMD.

We, therefore, devise a good strategy for setting the cut points which would work for many cases but not for all cases. In the following, we propose one such method which combines the first two steps listed previously into one. More specifically, the method constructs a path cover of *M*0 and also finds its equivalent path set in *M*1 hand in hand. We choose the cutpoints in any FSMD as follows.

1) The reset state is chosen.

2) A state *qi* is chosen if there is a divergence of flow from *qi*. More formally, *qi* is a cutpoint if $\exists c1, c2 \in S$ such that $c1 \neq c2$ and $<qi, c1, qj> \in f$ and $<qi, c2, ql> \in f$; *qj*, *ql* are not necessarily distinct.

Obviously, the cutpoints chosen by the aforementioned rules cut each loop of the FSMD in at least one cutpoint because each internal loop has an exit point. Corresponding to each path from a cutpoint to a cutpoint without having an intermediary cut point, we have to find an equivalent path in the other FSMD which, however, may not exist. Let p : [$q0i \Rightarrow q0j$] be such a path. The path is modified by concatenating with *p* all the paths from *q*0*j* to the subsequent cutpoints and trying to find their equivalent paths in the other FSMD. The process continues until a path of *M*1 that is equivalent to an extended path is obtained or the extension needs to be carried beyond the reset state or the extended path becomes a loop; in the last two cases, the algorithm reports the FSMDs to be possibly nonequivalent. This capability of the present method (of extending paths during equivalence checking) is central to its ability to handle situations where the path structure of the input behavior is changed by a path-based scheduler or an operation is moved beyond Basic Block boundaries. The verification algorithm is as follows:

Verification Algorithm

Input: The FSMDs *M*0 and *M*1.

Output: *P*0: a path cover of *M*0,

E: ordered pairs $<\beta, \alpha>$ of paths of *M*0 and *M*1, respectively, such that $\beta \in P0$ and $\beta \equiv \alpha$.

Step 1: Let η be the set of corresponding state pairs.

Let $\eta \leftarrow <q00, q10>$.

Insert cutpoints in *M*0 using the rule stated in the last section.

Let *P'*0 be the set of all paths of *M*0 from a cutpoint to a cutpoint having no intermediary cutpoint.

Let $P0$ and E be empty.
Step 2: If $P'0$ = *empty*, then return $P0$ as a path cover of $M0$ and E as the set of ordered pairs of equivalent paths of $M0$ (from $P0$) and $M1$ and *exit (success)*;
else go to Step 3.
Step 3: Find a path of the form $<q0i \Rightarrow q0f>$ from $P'0$ s.t. $q0i$ has a corresponding state $q1j$.
If no path is obtained, then go to Step 4;
else go to Step 5.
Step 4: If $P'0 \neq$ *empty*, then report "*$M0$ may not be contained in $M1$*" and *exit (failure)*;
else return $P0$ as a path cover of $M0$ and E as a set of ordered pairs of equivalent paths of $M0$ (from $P0$) and $M1$ and *exit (success)*.
Step 5: Let the path obtained in Step 3 be $\beta = <q0i \Rightarrow q0f>$.
Let $<q0i,\ q1j>$ be the corresponding state pair in η.
If $R\beta$ or $r\beta$ is undefined, then report "The $R\beta$ and/or $r\beta$ of β is not defined and exit (failure),"
else find a path of $M1$ emanating from $q1j$ which is equivalent to the path β. If such a path is found, then go to Step 6;
else go to Step 7.
Step 6: Let this path of $M1$ be α.
$\eta \leftarrow \eta \cup \{<endState(\beta),\ endState(\alpha)>\}$,
$E \leftarrow E \cup \{<\beta, \alpha>\}$,
$P0 \leftarrow P0 \cup \{\beta\}$,
$P'0 \leftarrow P'0 - \{\beta\}$.
go to Step 2.
Step 7: $P'0 \leftarrow P'0 - \{\beta\}$.
Extend β $(= <q0i \Rightarrow q0f>)$ in$M0$ by moving through the cutpoint $q0f$ until the next cutpoints but without moving through the reset state or any cutpoint more than once.
Let Bm be the set of all such extensions of the path β.
$P'0 \leftarrow P'0$—{paths of $P'0$ originating from $q0f$ which got appended to β}.
$P'0 \leftarrow P'0 \cup Bm$.
go to Step 8.
Step 8: If Bm = *empty*, then report "*β may not have any equivalent* in $M1$ *and cannot be extended*" and *exit (failure)*;
else go to Step 2.

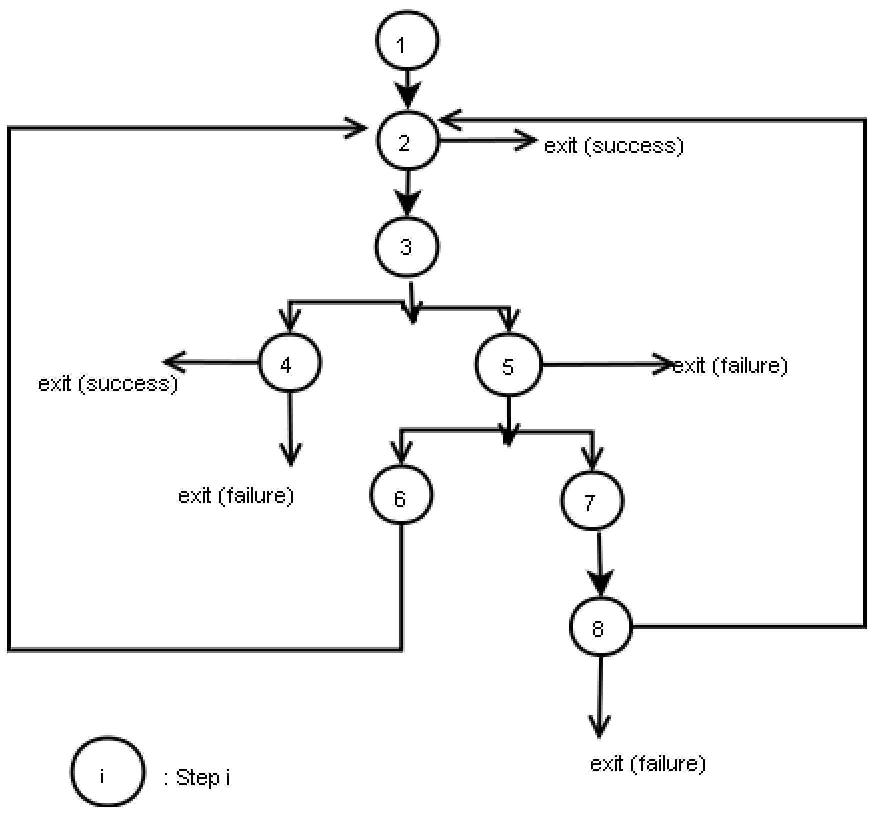


Fig. control flow of the verification algorithm in terms of steps.

The control flow among the steps of the algorithm is shown in above Fig. for clarity. The algorithm examines whether $M0$ equivalent to $M1$. In order to establish the computational equivalence between $M0$ and $M1$, the aforementioned algorithm is rerun with $M0$, $M1$ interchanged to determine whether $M1 \sqsubseteq M0$ or not.

Chapter 3

NORMALIZATION

3.1 Introduction:

While finding the equivalent path for a path, it is required to check the equivalence of the respective conditions as well as the data transformations of the paths. Since the condition of execution and the data transformation of a path involve the whole of integer arithmetic, checking equivalence of paths reduces to the validity problem of first order logic which is undecidable; thus, a canonical form does not exist for integer arithmetic.

The normalization process reduces many computationally equivalent formulas syntactically identical as it forces all the formulas to follow a uniform structure. In the following, the normal form chosen for the formulas and the simplification carried out on the normal form during the normalization phase are briefly described.

A condition of execution (formula) of a path is a conjunction of relational and Boolean literals. A Boolean literal is a Boolean variable or its negation. A relational literal is an arithmetic relation of the form $s\ R\ 0$, where s is a normalized sum and R belongs to $\{<=, >=, ==, \neq\}$. The relation $>$ $(<)$ can be reduced to $>=$ $(<=)$ over integers. For example, $x - y > 0$ can be reduced to $x - (y - 1) >= 0$.

The data transformation of a path is an ordered tuple $< ei >$ of algebraic expressions such that the expression ei represents the value of the variable vi after execution of the path in terms of the initial data state. So, each arithmetic expression in data transformation can be represented in the Normalized Sum form. A normalized sum is a sum of terms with at least one constant term; each term is a product of primaries with a non-zero constant primary; each primary is a storage variable, an input variable or of the form $abs(s)$, $mod(s1, s2)$, $exp(s1, s2)$ or $div(s1, s2)$, where $s, s1, s2$ are Normalized Sums. These syntactic entities are defined by means of production of the following grammar.

3.1 Grammar of the Normalized Sum:

1) $S \rightarrow S + T | c_s$, where c_s is an integer.
2) $T \rightarrow T * P | c_t$, where c_t is an integer.
3) $P \rightarrow abs(S) \mid (S)\ mod\ (S) \mid S \div C_d \mid v \mid c_m$, where $v \in I\ U\ V$, and c_m is an integer.
4) $C_d \rightarrow S \div C_d \mid S$.

Thus, the exponentiation and the (integer) division are depicted by infix notation and all functions have arguments in the form of normalized sums. In addition to

the above structure, any normalized sum is arranged by lexicographic ordering of its constituent sub expressions from the bottom-most level, i.e., from the level of simple primaries.
Example: The expression ($x + 3 * y + 7 >= 0$ && $4 * x^2 + 3 * y * z \neq 0$) will have the normal form [$1 * x + 3 * y + 7 >= 0$ && $4 * x * x + 3 * y * z + 2 \neq 0$].

3.2 Various simplifications that can be carried out at the normalization phase are as follows:

3.2.1) ***Simplification at the arithmetic expression (normalized sum) level:***

- Any expression involving only integer constants is immediately evaluated, e.g., $(5 / 2)$ is evaluated to 2.
- In an expression, common sub expressions are collected together. For Example: ($x^2 + 3 * x + 5 * z + 4 * x$) is reduced to ($1 * x * x + 7 * x + 5 * z$).

3.2.2) ***Simplifications at the relational expression (relational literal) level:***

- Any relational expression built from constant arithmetic expressions may be immediately evaluated to TRUE or FALSE. For example, $4 – 1 \ >= 0$ is evaluated to TRUE.
- Common constant factors are extracted from the normalized sum and the relational expression is consequently simplified. For example, ($3 * x * x + 9 * x * y + 6 * z + 7 >= 0$) is mapped to ($1 * x * x + 3 * x * y + 2 * z + 2 >= 0$), where $[7 \div 3] = 2$.

3.2.3) ***Simplification at the formula level:***

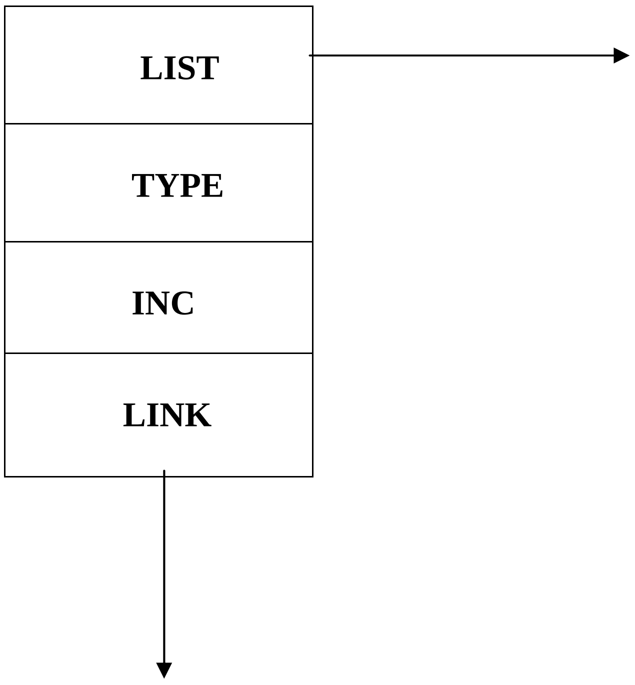
Some literals of the formula can be deleted by the rule " if (*A* --> *B*) then (*A* && *B* is equivalent to *A*) ". For this step of simplification, it becomes necessary to detect implication among literals. It is possible to detect whether a relational literal implies another relational literal when they involve the same non-constant sums. Let the literals be $l1: (s1 + c1) R1\ 0$ and $l2 : (s2 + c2) R2\ 0$. If $s1 == s2 == s$, then table depicts the relationship between the constants $c1$ and $c2$ depending upon $R1$ and $R2$, which must be satisfied for $l1$ to imply $l2$. Removal of repetitions of literals in a formula is possible using this rule as for any literal $l1$, $l1 \rightarrow l2$ is always TRUE. For example, the literal (*A* >= *B*) has multiple occurrences in the formula (*A* >= *B* && *C* <= *D* && *A* >= *B*). So, this formula is simplified to (*A* >= *B* && *C* <= *D*).

R1 \ R2	=	>=	≠	<=
=	*c1 = c2*	*c2 >= c1*	*c1 ≠ c2*	*c2 <= c1*
>=	-	*c2 >= c1*	*c2 > c1*	-
≠	-	-	*c1 = c2*	-
<=	-	-	*c2 < c1*	*c2 <= c1*

Table: Conditions on *c1* and *c2* for which *(s1 + c1)R1 0* implies (*s2 + c2)R2 0*

3.3 Data Structure for Normalized Form (A Normalized Cell) :

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



3.4 Representation of the Normalized Expressions :

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

1) A **LIST**-pointer , which points to the entries at the same level of the tree or equivalently, at the same hierarchal level of an expression.
2) A **TYPE**-field which indicates the type of the cell. Some typical examples of the types are ‘S’ for normalized sum, ‘T’ for normalized term, ‘R’ for relational literal, etc. TYPE = ‘v’ indicates a program variable or more generally a symbolic constant.
3) An integer field **INC** , the meaning of which varies from type to type. For example, TYPE = ‘S’, INC = 4 means that the integer constant in the normalized sum is 4.
4) A **LINK**-pointer , which points to the leftmost successor of the node in question in the next level of the tree or equivalently, in the next syntactic level of the expression.

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

3.4.1 Normalized sum:

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

Normalized sum: $3 + 2 * a + 5 * x * y$.

Example:

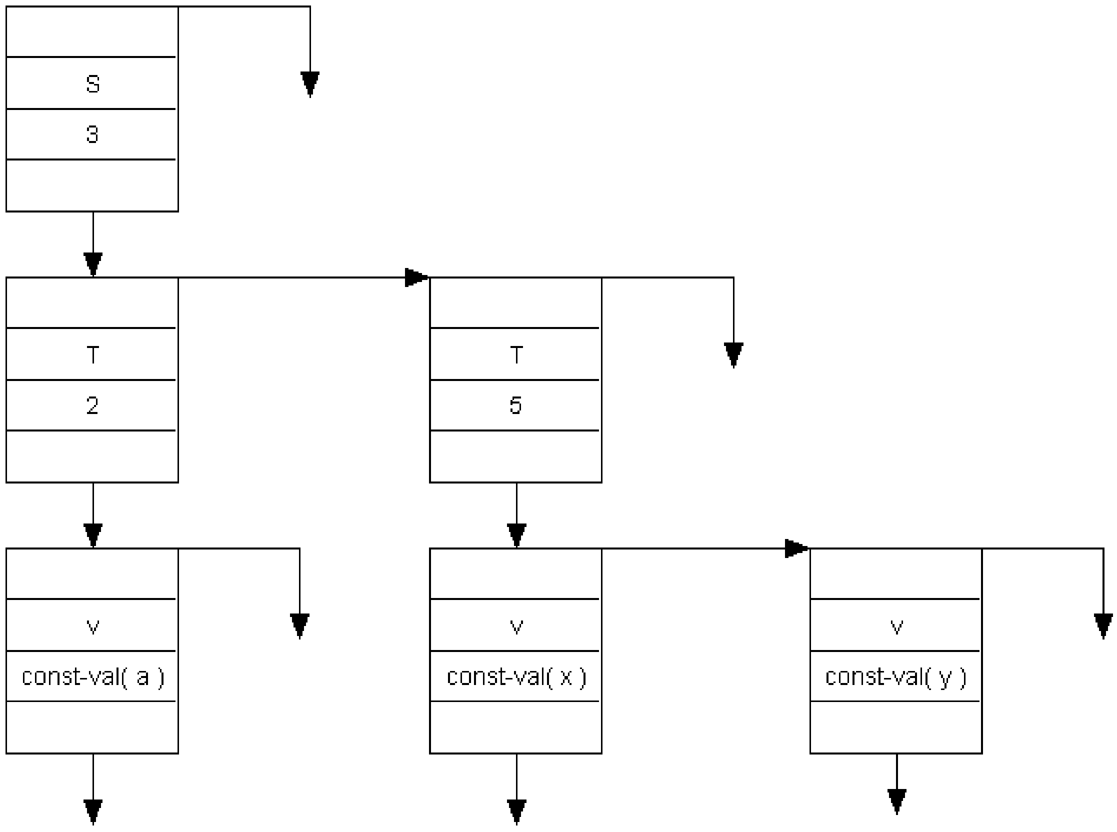
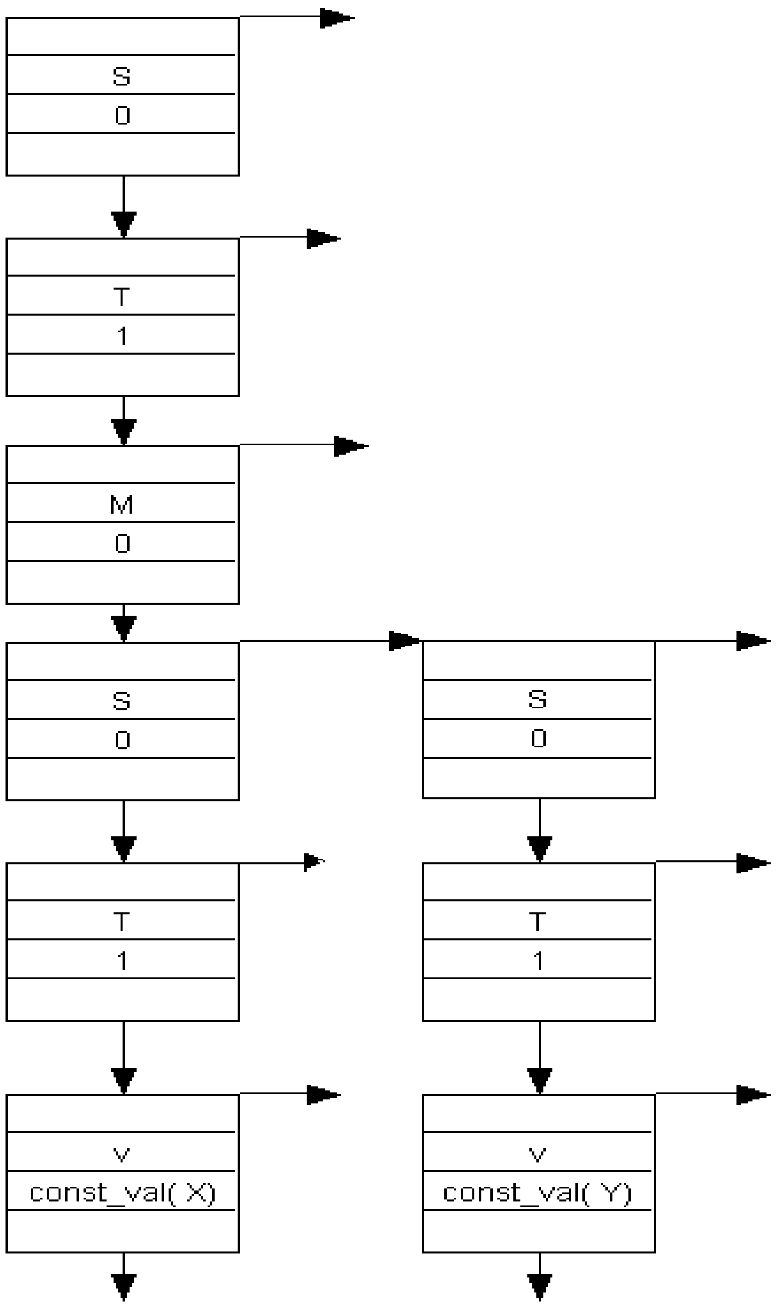


figure - Normalized sum for : 3 + 2 * a + 5 * x * y

3.4.2 Representation of a MOD or DIV Expression:

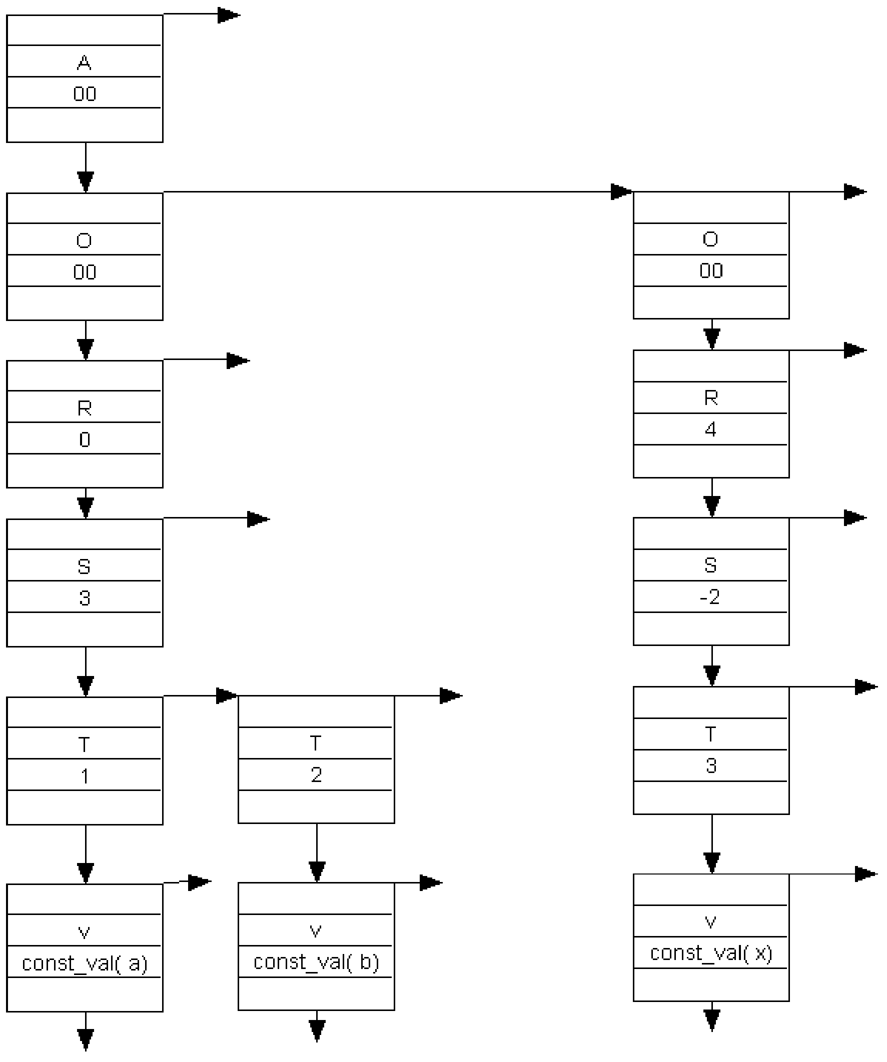
x % y = 0 + 1 * (M 0 0 + 1 * x, 0 + 1 * y)



3.4.3 Normalized Condition:

A Normalized Condition is represented as *[(s + c) R 0],* where *'s'* is a Normalized Sum, *'c'* is an integer constant, *'R'* is a Relational operator. Example: Normalized Condition: *[(3 + 1 * x – 2 * y) >= 0].*

An Example:



CONDITIONAL EXPRESSION

[(s1 + c1) R1 0] AND [(s2 + c2) R2 0] => [(3 + 1 * a - 2 *b) >= 0] AND [(-3 + 1 * x) == 0]

3.5. NORMALIZATION ROUTINES:

The normalizer is a hierarchically organized module consisting of routines at various levels. In order to accomplish a particular task assigned to a routine at some level, it can take the help of any other routine, which is at a lower level routine, but a lower level routine cannot invoke the higher level ones. All the routines assume the inputs to be in normal forms and produce the output in normal form.

3.5.1 Remove all multiple occurrences of normalized conditions in a Conditional expression:

This function removes the multiple occurrences of conditions from the conditional expression and returns the resultant conditional expression.
Example:
Input: Conditional Expression = *(A && B && B && B && A).*
Output: Resultant Conditional Expression (*A && B).*

Where, A and B are conditional expressions.

3.5.2 Is Condition A Implies Condition B:

This function checks if condition *A* implies condition *B* in a Conditional Expression *"A && B*". If success, then Condition *B* is deleted from Conditional Expression and the resultant conditional expression (*A*) is returned.
Example:
Input: *[(-6 + 2 * x) >= 0] && [(3 + 2 * x) >= 0]*
Processing: Let *A = [(-6 + 2 * x) >= 0],* and *B = [(3 + 2 * x) >= 0]*
Since *A* implies *B*, *B* will be deleted from Conditional Expression
Output: *[(-6 + 2 * x) >= 0]*

3.5.3 Simplify Sub Expressions in Sum:

This function does the sub expression simplification of a normalized sum. The common sub expressions are collected together. It returns the resultant normalized sum.
Example:
Input Sum: *5 + 2 * x + 3 * x + 1* z*
Output Sum: *5 + 5 * x + 1 * z*

3.5.4 Simplify Sub Expressions in a Condition:

This function does the sub expression simplification of a normalized condition. It in turn calls the function "**Simplify Sub Expressions in Sum**" and the common sub expressions of the normalized sum are collected together and returns the resultant normalized condition.
Example:

Input Sum: *[(5 + 2 * x + 3 * x + 1* z) >= 0]*
Output Sum: *[(5 + 5 * x + 1 * z) >= 0]*

3.5.5 Arithmetic Simplification of a Conditional Expression:

This function does the arithmetic simplification of the normalized sum of a conditional expression. The common constant factors are extracted from the normalized sum.
Example:
Input sum: *7 + 3 * a + 9 * x.*
Output sum: *2 + 1 * a + 3 * x*

3.5.6 Substitute a Primary In a Normalized Sum:

This function works on the normalized sum *'s'* to substitute each occurrence of primary *'x'* in *'s'* with the normalized sum *'z'*.
Example:
Input: *s = 3 + 2 * x*, Primary = *x*, *z = 3 + 1 * p*.
Output: resultant Normalized Sum = *9 + 2 * p*.

Chapter 4

CODE MOTION VERIFICATION

4.1 Introduction:

It may be possible to transform the input behavior to some equivalent description which results in amore efficient scheduled behavior. This fact underlines the need for incorporating high-level code-transformation techniques in the scheduling phase of synthesis to overcome the effects of programming style on the quality of generated circuits. Needless to say, these transformations increase the scheduling verification challenges.

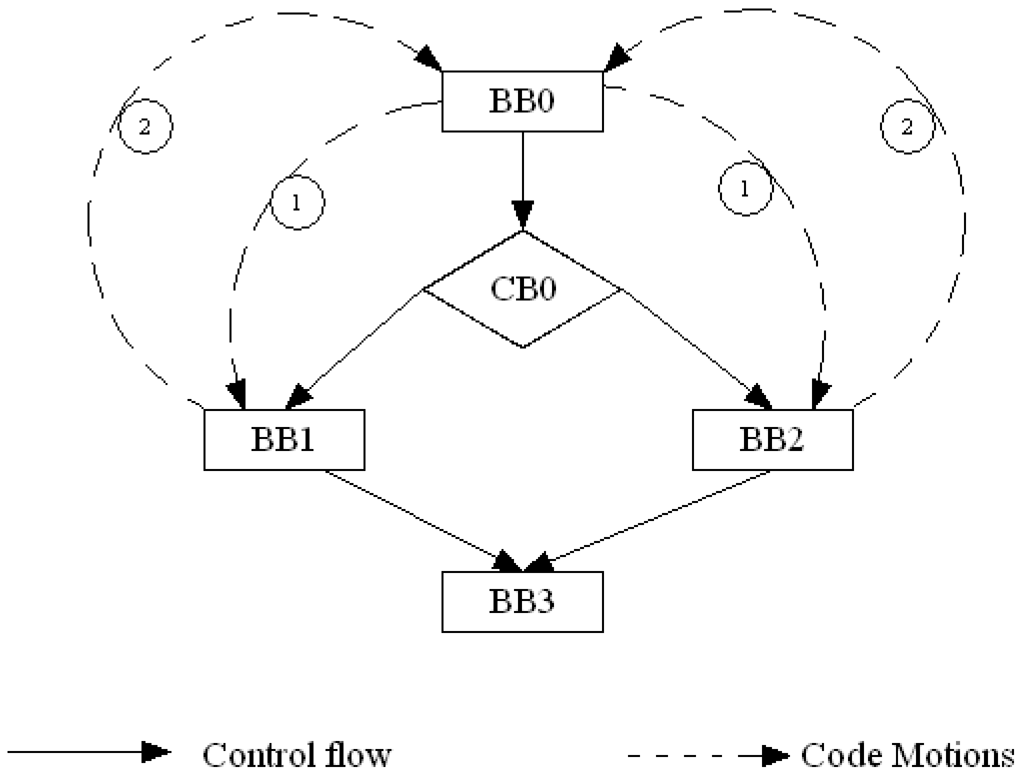


Fig. Various Code-Motion techniques.

4.2 Code-Motion Techniques:

4.2.1 Duplicating Down: It moves operations from a Basic Block (BB) preceding a Conditional Block (CB) to both the succeeding BBs. This is shown by the arcs marked 1 in fig. Reverse Speculation and early Condition execution belong to this category.

4.2.1.1 Reverse Speculation: In reverse speculation the operations before a conditional Block are moved into the blocks subsequent to the conditional block In a special case of Reverse Speculation the scheduler may move an operation, say '*O*', before the conditional block into only one of the conditional branch. This is possible when the operations in the other branch as well as all the operations following the merging of the conditional branches are not dependent on the result of the operation '*O*'.

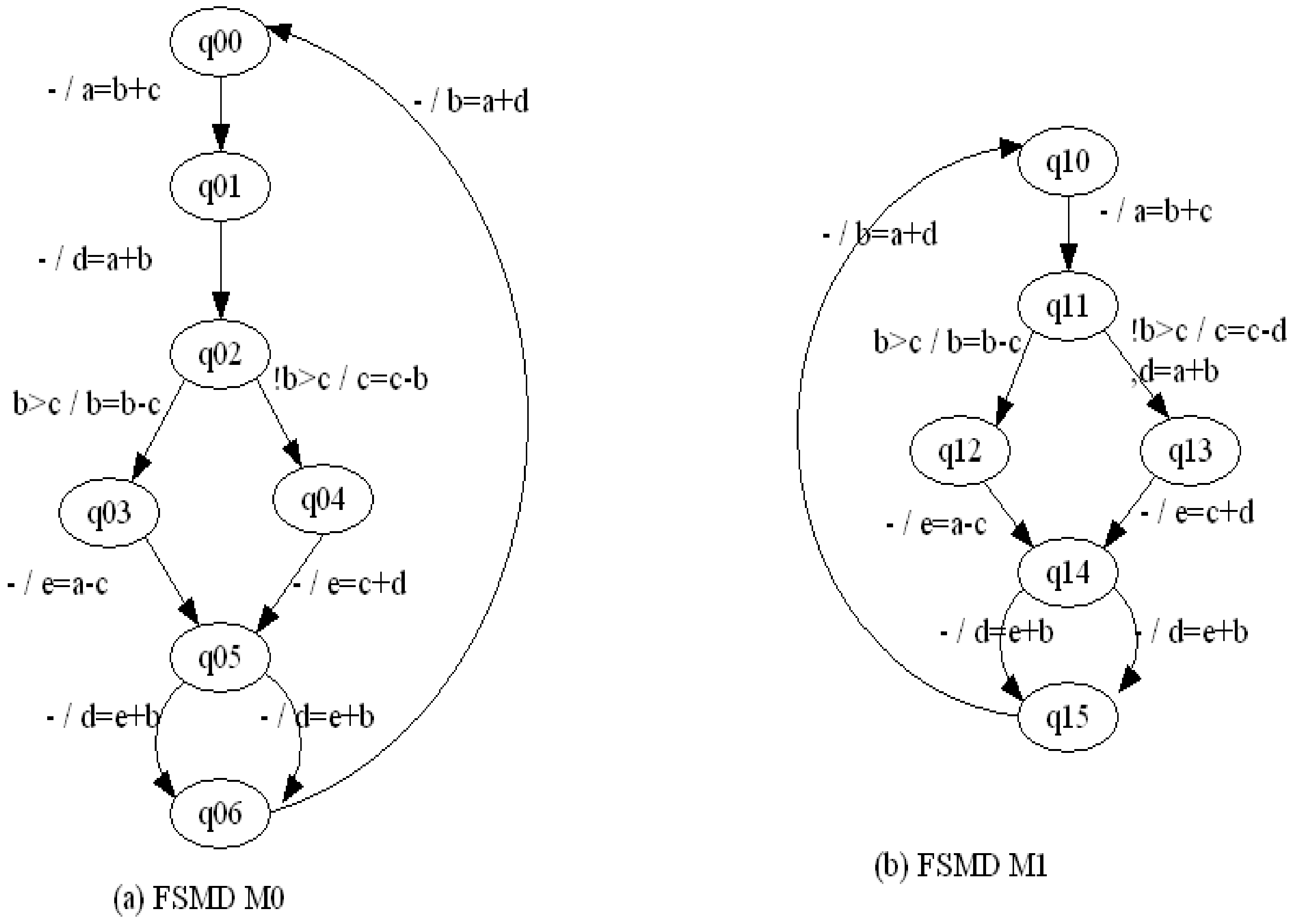


Fig. Reverse Speculation

Consider the example in Fig. The operation *`d <= a + b'* of the original behavior (FSMD *M0*)is moved to only one conditional branch with condition *'!b > c' in the FSMD M1*. This is possible because the operations in the conditional branch with condition *'b > c'* and all possible execution paths following the merging node *q0i5* of *M0* do not use the value of *d* which is *a + b*. The

Verification algorithm fails in this case. However, one modification in verification algorithm will suffice for equivalence checking.

Let 'β' be a path in *M0* of the form < *q0i* => *q0j* > and < *q0i, q1k* > be a corresponding state pair. When the existing algorithm fails to find the equivalent path for 'β', then let there exist a path starting from *q1k* in *M1*, say 'α', whose condition of execution matches with that of 'β' but the data transformations does not match. In such case, we will check whether there is any variable of *'V0 ∩ V1'* (*V0* contains the list of variables present in FSMD *M0*), and *V1* contains the list of variables present in FSMD *M1*) which is modified along 'β' but not modified in the path 'α'. Let '*v*' be such a variable. Without any loss of generality, let the values of all the variables in *'V0 ∩ V1'* other than *'v'* at the end of execution of 'α' be the same as those for 'β'. Now, if we can show that the transformed value of '*v*' in 'β' is not used in any execution path starting from state '*q0j*', then 'α' is equivalent to 'β' even if their respective data transformations match partially (only on the other variables). In other words, if the variable *'v*' is always used only after it is defined in the subsequent execution paths from *q0j*, then there is no use of the operation that updates *'v'* in β and we remove this operation during scheduling.

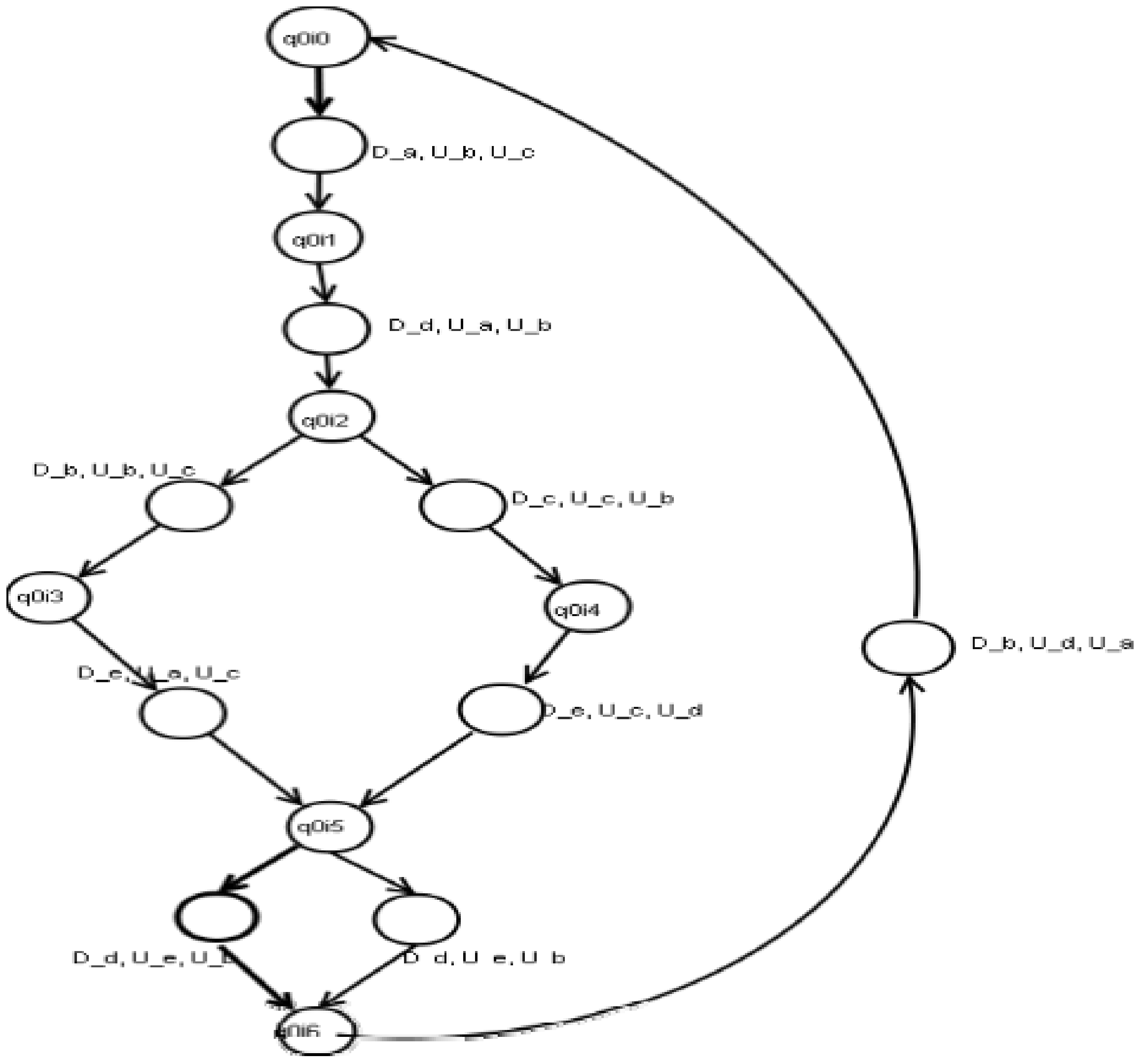


Fig. Kripke structure for FSMD *M0*.

We convert the FSMD *M0* into an equivalent Kripke structure by some logical transformations (Fig.). A dummy state will be added for every transition of *M0*. There would be two propositions, *'D_v'* (defined *'v'*) and *'U_v'* (used *'v'*), for each variable in *'V0 ∩ V1'*. The proposition *'D_v'* will be true in a dummy state if the variable *'v'* is defined by some operation in the corresponding transition in the *M0*. Similarly, *'U_v'* will be true in a dummy state if the variable *'v'* is used in some operation in the corresponding transition in *M0*. By convention if any proposition is not present in any state of the Kripke structure, then the negation of the proposition is true in that state. The required property that there does not exist any path in which *'v'* has not been defined before it is used can be written as the CTL formula " ! E [(! *D_v*) W *U_v*] ", were E represents there exists and W represents weak until operator. This formula can be verified using CTL model checker ex. NuSMV. If this formula is true in the state *'q0j'*, then 'β' is equivalent to 'α'.

Consider the example in Fig. The algorithm considers $\beta = q0i0 \rightarrow q0i1 \rightarrow q0i2$ of *M*0 and fails to find the equivalent path in *M*1 in step 5. It finds $\alpha = q1j0 \rightarrow q1j1$ in *M*1 which has the same condition (true) as that of β but the variable *d* is transformed along β but not along α; the other variable *a* gets transformed identically. It, next, finds that the formula " ! E [(! *D_v*) W *U_v*] " is not true in state *q0i2* in the Kripke structure of the FSMD *M*0. So, the control goes to step 7 and extends β. The extended paths are $\beta = q0i0 \rightarrow q0i1 \rightarrow q0i2 \xrightarrow{b > c} q0i3 \rightarrow q0i5 \rightarrow q0i6 \rightarrow q0i0$ and $q0i0 \rightarrow q0i1 \rightarrow q0i2 \xrightarrow{!b > c} q0i4 \rightarrow q0i5 \rightarrow q0i6 \rightarrow q0i0$. The equivalent path of the latter one in *M*1 is $q1j0 \rightarrow q1j1 \xrightarrow{!b > c} q1j3 \rightarrow q1j4 \rightarrow q1j5 \rightarrow q1j0$. Step 5 fails to find the equivalent path of the former path; it then finds the path $\alpha = q1j0 \rightarrow q1j1 \xrightarrow{b > c} q1j2 \rightarrow q1j4 \rightarrow q1j5 \rightarrow q1j0$ in *M*1 which has the same condition of execution with β. Again, the variable *d* is transformed along β but not along α. It, next, finds that the formula " ! E [(! *D_v*) W *U_v*] " is true in state *q0i0* in the Kripke structure of the FSMD *M*0. So, α is equivalent to β. In this way, the equivalence of *M*0 and *M*1 can be established.

4.2.1.2 Early Condition Execution: This transformation involves restructuring the original code so as to execute the conditional operations as soon as possible. This, in effect, means that the conditional operation is moved-up in the design, and hence, all the operations before the conditional operation are reverse speculated into the conditional branches.

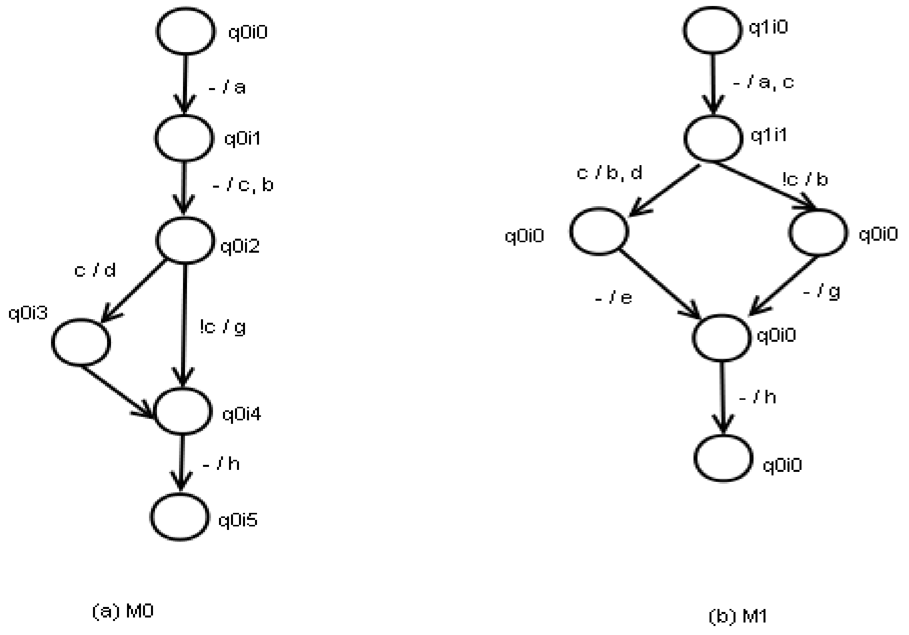


Fig. Early Condition Execution

In the above Fig. the conditional statement operation *'c'* is executed one step early in the scheduled behavior and the operation *'b'* is reverse speculated in the conditional branches. This is also a kind of Reverse Speculation and can be handled.

4.2.2 Boosting Up: It moves operations from a Basic Block (BB) within a conditional branch to its preceding BB. This category is shown as arcs 3 in fig. The code-motion techniques, like speculation and loop shifting, fall under this category.

4.2.2.1 Speculation: Speculation refers to the unconditional execution of instructions that were originally supposed to be executed conditionally. In this approach, the result of a Speculated operation is stored in a new variable. If the condition under which the operation was to execute evaluates to true, then the stored result is copied to the variable from the original operation, else the stored result is discarded.

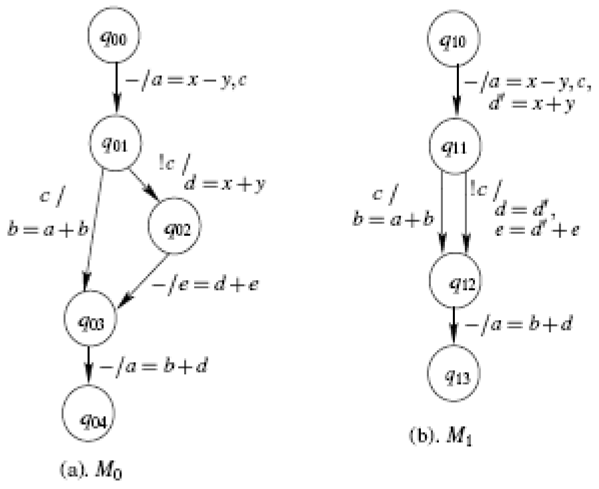


Fig. Speculation

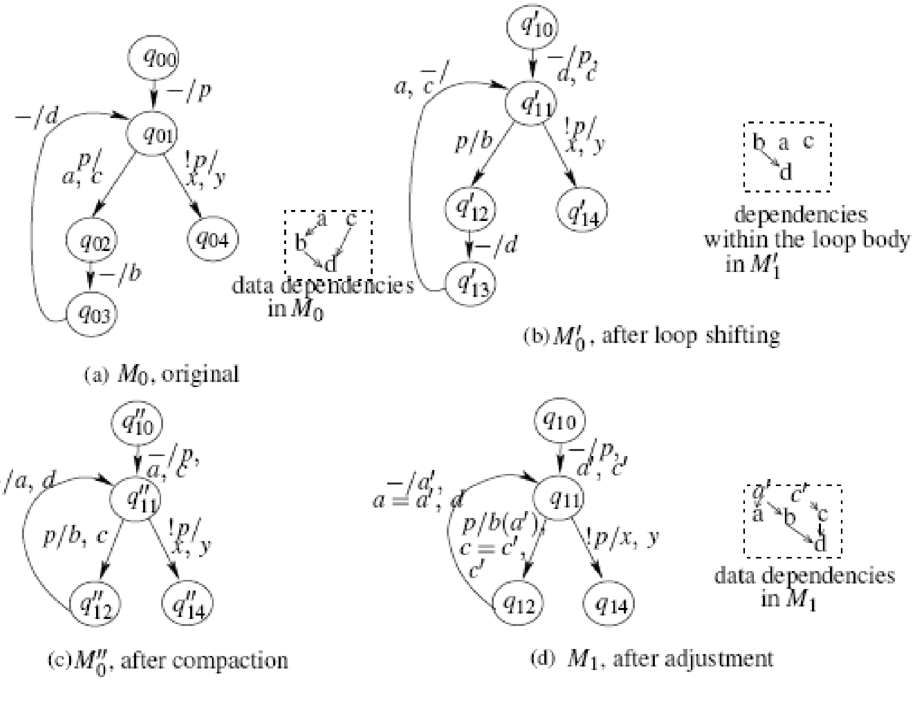
In the above figure, the operation *d=x+y* is speculated out of the branch with condition *'!c'* of the FSMD *M0* and the result of the operation is stored in *d'*. It may be noted that if we do not store the value in *d'*, then the variable *'a'* gets the wrong value (by the operation *a=b+d*) when the execution is through the branch with condition *'c'* of the FSMD *M1*.

The proposed Verification algorithm fails in this case. The algorithm uses any node with more than one outward transition as a cutpoint. So the node $q0i1$ in $M0$ of Fig. is a cutpoint. The algorithm first tries to find an equivalent of the path $\beta = q0i0 \rightarrow q0i1$. It finds $\alpha = q1j0 \rightarrow q1j1$ as the equivalent path of β because the condition of execution (TRUE) and the data transformation of the common variables (*a, b, c, d, e, x and y*) are the same for β and α. Next, the algorithm tries to find the equivalent of the path $\beta = q0i1 \xrightarrow{c} q0i3 \rightarrow q0i4$. The corresponding equivalent path found by the algorithm is $\alpha = q1j1 \xrightarrow{!c} q1j2 \rightarrow q1j3$. Now, it tries to find the equivalent of the path $\beta = q0i1 \xrightarrow{!c} q0i2 \rightarrow q0i3 \rightarrow q0i4$. The algorithm fails to find the equivalent path of β because the data transformation of β is $<< b + x + y,\ b,\ c,\ x + y,\ x + y + e,\ x, y>, - >$ where the variables are in the order $a < b < c < d < e < x < y$. There is no path in $M1$ starting from $q1j1$ with the same data transformation.

Actually, the path $\alpha = q1j1 \xrightarrow{!c} q1j2 \rightarrow q1j3$ of $M1$ is equivalent to the path β. As we are using *symbolic simulation* to find the data transformation, the final values of *d, a* and *e* in α will be in terms of *d'*. Specifically, the data transformation of α is $<< b + d',\ b,\ c,\ d',\ d' + e,\ \ x,\ \ y,\ d'>, - >$, where the variables are in the order $a < b < c < d < e < x < y < d'$ and will not match that of β. But if we use the right hand side expression $x + y$ of the operation $d' <= x + y$ that defines *d'* in the path $q1j0 \rightarrow q1j1$ as the initial symbolic value of *d',* then the data transformation in the path α becomes $<< b + x + y,\ b,\ c,\ x + y,\ x + y + e,\ x,\ y>, - >$ which is equal to that of β. Thus, α can be ascertained to be equivalent to β.

The following simple modifications in the proposed verification algorithm can handle this code motion technique. We put cutpoints in $M1$ by the rules proposed and find the set *P'1* of paths from one cutpoint to another without having any intermediate cutpoints. While finding the equivalent of a path, say β, of $M0$ in $M1$, paths starting from the corresponding state of the start node of β are considered one by one. Let β be of the form $< q0i \rightarrow q0j >$ and $< q0i,\ \ q1k >$ be the corresponding state pair. So, the paths starting from $q1k$ will be checked one by one until an equivalent path is found, failing which it is concluded that no equivalent path exists for that path. Let α be a path which starts from $q1k$. If it is found that the some variables not belonging to $V0 \cap V1$ are used before they are defined along α during computation of $R\alpha$ and $r\alpha$, then we will find the set of paths from *P'1* which terminate in $q1k$. Next, the last operations defining these variables in those paths will be found out. This can be done by *backward breadth first search* from $q1k$. The right hand side expressions of those operations will be used as the initial symbolic values of these variables. Consider, for example, the path $q0i1 \xrightarrow{!c} q0i2 \rightarrow q0i3 \rightarrow q0i4$ as β in Fig. The state $q1j1$ is the corresponding state of $q0i1$. Consider the path $q1j1 \xrightarrow{!c} q1j2 \rightarrow q1j3$ as α. The variable *d'* (not belonging to $V0 \cap V1$) is used (in the operation $d <= d'$) before it is defined along β. Now, the paths $q1j0 \rightarrow q1j1$ is the only path which terminates in $q1j1$ and the operation $d' <= x + y$ defines *d'* in that path. So, the expression $x + y$ will be used as the initial symbolic value of *d'* while computing the condition of execution and the data transformation of α. With this modification, the path α will be found as the equivalent path of β. However, the initial symbolic value obtained for every variable should be unique in all the paths that terminate in $q1k$. If more than one expression are found for a particular variable or no operation is found which defines a variable in one path, we will ignore a for equivalence checking and consider the next path from $q1k$.

4.2.2.2 Loop Shifting and Compaction: *Loop shifting* is a technique whereby an operation *op* is moved from the beginning of the loop body to the end of the loop body. To preserve the correctness of the program, a copy op_c of the operation *op* is also placed before the start of the loop. Consider the example in Fig. Operations *a* and *c* of the original behavior are shifted to the end of the loop body as well as placed at the entry edge(s) of the loop. The modified FSMD is shown in Fig (b).



(a) M_0, original
data dependencies in M_0
(b) M'_0, after loop shifting
dependencies within the loop body in M'_1
(c) M''_0, after compaction
(d) M_1, after adjustment
data dependencies in M_1

Fig. Loop Shifting and Compaction

It is important to note that shifting operation(s) in a loop reduces the data dependencies among the operations within the loop body. For example, if we consider the *ith* iteration of the loop in the original behavior in the FSMD *M*0 of Fig (a), the value of the variable *b* depends on *a* and the value of *d* depends on *c*. But, if we consider the *ith* iteration of the loop in *M'0*, then the operations *a* and *c* of the loop body represent the operations corresponding to the *(i + 1)th* iteration of *M*0. So, the operations *b* and *d* in the loop body do not depend on the operations *a* and *c*, respectively. The modified data dependency in the shifted loop is shown in the dotted block in Fig (b). As a result, the scope of concurrent execution of the operations increases within the loop body. Specifically, it is possible to execute the operations *b* concurrently with *a* and the operation *c* concurrently with *d*. The compacted FSMD is shown in Fig. (c).

Shifting an operation results in execution of the operation one more time than in the original code. For example, if we consider *n* number of iterations of the loop body, then the operations *a* and *c* execute $n + 1$ times in the compacted FSMD (Fig. (c)) whereas they would execute *n* times in the original description (Fig. (a)). It needs to ensure that executing the shifted operation one extra time does not change the behavior of the program. In order to do so, it is required to perform the following steps. Let the shifted operation be $v \leftarrow f(\,)$. The first step is to create an operation "$w \leftarrow f(\,)$" in place of the shifted operation, where *w* is a new variable. In the second step, all the instances of the operation $v \leftarrow f(\,)$ in the loop body in the original behavior are replaced by two operations " $v \leftarrow w$" and "$w \leftarrow f(\,)$" in parallel. Finally, the operations that use the variable `*v*' in the loop body will now use the variable *w* instead of variable *v*. It is demonstrated in Fig. (d). The results of the shifted operations *a* and *c* are stored in *a*' and *c*', respectively. The operation *a* is replaced by $a <= a'$ and *a*' in the loop body. Thus, by the *ith* iteration, *a*' is computed $i + 1$ times but *a* assumes the value of *ith* iteration. Similarly, the operation *c* is also replaced. In the original behavior, the operation *b* uses the variable *a* and the operation *d* uses the variable *c*. Hence, they will now use the variable *a'* and *c'*, respectively. In the Fig. (d), *b(a*') indicates that the operation *b* uses the variable *a*'. Similarly, *d(c')* indicates that the operation *d* uses the variable *c*'.

This situation is similar to the speculation above subsection. So, the proposed modification of the algorithm in that subsection can handle this situation. Here, we need to check the equivalence between the FSMDs $M0$ in Fig. (a) and $M1$ in Fig. (d). For the paths $q0i0 \rightarrow q0i1$ and $q0i1 \ {}_{\mathrm{P}}\!\!\rightarrow q0i4$ in $M0$, $q1j0 \rightarrow q1j1$ and $q1j1 \ {}_{\mathrm{P}}\!\!\rightarrow q1j4$, respectively, are ascertained to be the equivalent paths in $M1$. Now, while finding the equivalent path of $\beta = q0i1 \ {}_{\mathrm{P}}\!\!\rightarrow q0i2 \rightarrow q0i3 \rightarrow q0i1$, it was found that the variables *a', c'* ($\in V1 - V0$) are used before they are defined along $\alpha = q1j1 \ {}_{\mathrm{P}}\!\!\rightarrow q1j2 \rightarrow q1j1$. Hence, we need to find the paths that terminate in the state $q1j1$. The paths $q1j0 \rightarrow q1j1$ and $q1j1 \ {}_{\mathrm{P}}\!\!\rightarrow q1j2 \rightarrow q1j1$ are such paths. In both the paths, the same operations *a*' and *c'* update the variables *a'* and *c'*, respectively. In fact, if the right hand side of the respective operations is used as the input assertions for *a*' and *c'*, then α will become the equivalent path of β.

Chapter 5

IMPLEMENTATION OF A TEST BENCH: (*IEEE754*)

5.1 "IEEE754 FLOATING POINT UNIT – ARITHMETIC OPERATIONS"

1) Addition of two Floating Point Numbers.
2) Subtraction of two Floating Point Numbers.
3) Multiplication of two Floating Point Numbers.
4) Division of two Floating Point Numbers.

INPUT: The Integer value of the 32-bit (*IEEE754*) representation of the two Floating Point Numbers.

PROCESSING: The Mantissa, the Exponent, and the sign of the two Floating Point numbers are extracted from the integer values using some integer arithmetic operations (without the use of arrays).

SIGN = (INPUT / 2 ^ 31) % 2

MANTISSA = INPUT % (2 ^ 23) + (2 ^ 23)

EXPONENT = (INPUT * 2) / (2 ^ 24)

31	30							24	23																						0

SIGN = 31ST bit

EXPONENT = 24 – 30 (8-bit)

MANTISSA = 0 – 23 (24-bit)

OUTPUT: The integer value of the 32-bit (*IEEE754*) representation of the resultant floating point number.

RESULT = SIGN * (2 ^ 31) + EXPONENT * (2 ^ 23) + MANTISSA % (2 ^ 23)

6. REFERENCES:

[1] D. Sarkar and S. C. De Sarkar, ***"Some Inference Rules for Integer Arithmetic for Verification of Flowchart Programs on Integers",*** IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, VOL. 15**,** NO. JANUARY 1989.
[2] D. Sarkar and S. C. De Sarkar***, "A Theorem Prover for Verifying Iterative Programs Over Integers",*** IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, VOL. 15, NO. 12, DECEMBER 1989.
[3] Chandan Karfa, ***"Hand-in-hand verification and Synthesis of Digital circuits"***, M. S. Thesis, I.I.T. KHARAGPUR, 2007.
[4] Chandan Karfa, Chittaranjan Mandal, Dipankar Sarkar, Pramod Kumar, ***"An Equivalence Checking Method for Scheduling Verification in High – Level Synthesis"***, IEEE Transactions on COMPUTER-AIDED DESIGN of Integrated Circuits and Systems, VOL. 27, No. 3, March-2008.