A NEW APPROACH FOR THE SYNTHESIS OF FSM'S FROM CONTROL-FLOW GRAPHS

by

Masatoshi Sekine, Tiziano Villa, Kenji Goto,
and Robert K. Brayton

Memorandum No. UCB/ERL M93/59

28 July 1993

A new approach for the synthesis of FSM's from control-flow graphs

by

MASATOSHI SEKINE*1, TIZIANO VILLA*2, KENJI GOTO*3,

ROBERT K. BRAYTON*2

28 July 1993

*1:ULSI Laboratory, Toshiba Research Development Center,

*2:Electrical Engineering & Computer Science, UC at Berkeley,

*3:Ome Works, Toshiba Corpolation

ELECTRONICS RESEARCH LABORATORY

College of Engineering

University of California, Berkeley,

94720

A NEW APPROACH FOR THE SYNTHESIS OF FSM'S FROM CONTROL-FLOW GRAPHS

by

Masatoshi Sekine, Tiziano Villa, Kenji Goto,
and Robert K. Brayton

Memorandum No. UCB/ERL M93/59

28 July 1993

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

A NEW APPROACH FOR THE SYNTHESIS OF FSM'S FROM CONTROL-FLOW GRAPHS

by

Masatoshi Sekine, Tiziano Villa, Kenji Goto,
and Robert K. Brayton

Memorandum No. UCB/ERL M93/59

28 July 1993

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

A new approach for the synthesis of FSM's from control-flow graphs

by

MASATOSHI SEKINE*1, TIZIANO VILLA*2, KENJI GOTO*3,

ROBERT K. BRAYTON*2

28 July 1993

*1:ULSI Laboratory, Toshiba Research Development Center,

*2:Electrical Engineering & Computer Science, UC at Berkeley,

*3:Ome Works, Toshiba Corpolation

ELECTRONICS RESEARCH LABORATORY

College of Engineering

University of California, Berkeley,

94720

A new approach for the synthesis of FSM's from control-flow graphs

by

MASATOSHI SEKINE*1, TIZIANO VILLA*2, KENJI GOTO*3, ROBERT K. BRAYTON*2

ELECTRONICS RESEARCH LABORATORY

College of Engineering

University of California, Berkeley,

94720

ABSTRACT

This paper proposes a new approach based on loop constructs for the derivation of finite state machines (FSM's) from high level models (HLM's), and also for state codes by using weights of eigenvectors of state transition matrices. As loops are closed and defined locally, they are primitive and robust under the transformation from HLM to FSM. The loops in control flow-graphs (CFG's) have been transformed to similar loops in FSM's in traditional approaches, but these approaches are valid only as far as control in CFG's is suitable to FSM's. Loops having complex control constructs over macro-sequences in CFG's must be reformed in order to control micro-sequences in FSM's. The loop body decomposition by Shannon expansion is proposed to obtain suitable micro-loops for FSMs. It generates recursive loop equations and micro-loop pieces from upper-level loop bodies. Since upper-level loops are designed to control global paths of data-flow parts, some lower level loops control global paths. These loops are desirable for global optimization. The global data-flow parts selected by such loops are ordered and optimized incrementally according to the weight order of the loops. Scheduling for each loop generates new control variables for the state transitions in it instead of variables occurring in control-flow parts of the HLM. A mathematical basis is built to relate loops in the state diagrams and eigen-vectors or poles in pulse transform functions. The relations between linear systems and Boolean circuits are derived by combining a vector space and logic operations. The new approach based on loop body decomposition provides no heuristic algorithm, but a deductive algorithm. It is shown that the weights of the states are obtained in a general form if the state transition is a linear group.

1.0 Introduction

Simple models are demanded for describing huge systems of tens of millions of gates in chips by the end of this century. Systems are modeled as collections of concurrent abstract modules: block, process, and function. It is necessary to formulate *specifications* with integer numbers, floating, or complex numbers in order to express mathematical objects(four elementary operations(+-x/), linear algebra, or differential equations). A key step in High Level Synthesis (HLS) is to map control flow graphs (CFG's) to finite state machines (FSM's). Some mappings are complex due to a large degree of freedom in searching optimum FSM's[1],[6]. Orders or priority weights of components are specified to exclude unnecessary searches for solutions in a design space. As there is no intrinsic ordering in Boolean equations, we introduce orderings in each design method. Each lower-level ordering must be deduced from upper-level orderings. State assignment programs for FSM's, such as NOVA[10], ASYL[7], and Mustang[11], have been developed and their effectiveness to optimize entire FSM's with two-level and/or multilevel implementation is well known. However, their implementations depend on the starting point. In general, FSM's are used to control data-path circuits, and control circuits depend on the data-path circuits controlled by them. On the other hand, in the high-level synthesis, some sophisticated methods for scheduling and control allocation have been developed such as the condition vector in the system Cyber[9], and relative control generation in the system Hebe[1]. They can operate on the complex structure of control-flow graphs and generate a reduced schedule. However, since they are based on the assumption that a control step is equivalent to a single state of a FSM, each control step is simply assigned to one FSM. We must examine whether such FSM is optimal. New approaches for utilizing regularity and symmetry in systems are reported. Amann and Baitinger[3] proposed new-state-assignment algorithms by using counters to implement the longest state transition chain. The state chains are coded so that a maximum of the remaining coding constraints are satisfied. Stok[4] discussed false loops through resource-sharing caused by both data-flow and control-flow. Both approaches used more complex objects than Boolean variables for handling large structures. Nourani and Papachristou[5] have introduced a Liapunov stability theorem using a transformation technique between the design space and the dynamic system space.

This paper describes a new approach for the synthesis of FSM's from control-flow graphs by introducing a loop body decomposition relation with respect to variables occurring more frequently in the control-flow. A new ordering is presented to distinguish loops from each other with respect to their weights. We treat the loops of state transitions to be primitive elements which are robust under transformations from high level models(HLM's) to FSM's. Then, we introduce linear algebra to describe them. The behavioral transformation algo-

rithm is realized with maps from one algebra to another. For this purpose we investigate useful algebras which calculate equations for loops at the HLS level as Boolean algebra does at the logic synthesis level. Some formulations are taken from Lie algebra.

2.0 Flow of the synthesis procedure

1. Hardware model:

Let a system be modeled with abstract components. Each component is described with control parts and data flow parts regardless of its functionality described by means of a declarative expression (called a block model) or of a procedural expression (called a process, a procedure, and a function model). There are two types of control-flow constructs: 1) branches {if-then-else, switch-case}, 2) loops {for-loop, while-do loop, repeat-until loop, and Do while loop}. The data-flow constructs are assignments, function calls, statements, and complex statements. A control-flow construct (Cu, Mu(Bk's, (Cv, Mv))) has a conditional expression Cu and an execution body Mu in which data-flow constructs Bk's or nested control parts (Cv, Mv) are described. Data entities (constants and variables) can be of control and data flow type . All the data entities appearing in conditional expressions of the control-flow are of control type, and the others are of data flow type.

As an example, consider a simple behavioral model BM written in the C language.

```
k=(icd==add); i=(icd==mul); j=(icd==sop); P=0; reset=0; ready=0;
while(Q){if(P'&reset) ready=1; if(ready&icd) P=1;
        if(P){if(k) s=b+c; else if (i) s=b*c;} if(P&j) s=b*c+d*e; if(error) P=0;}        ( 1 )
```

where "+" is an adder operator, "*" is a multiplier, and Cq=while(Q), Mq={if(P'..)..}, Br=(ready=1), Cp=if(p), Mp={if(k) s=...}, Bs_1=(s=b+c). The variable Q, reset, ready, and error are external. The other external variable icd stands for a command that specifies the instructions of addition, multiplication, and sum of multiplications. The k, i, and j are logical variables defined by conditional expression of an equal operator "==". The top loop construct while(Q) is necessary for BM to run permanently. The initialization of BM is done successively by the first two expressions in the while loop. The variable P defines the execution mode of BM, and is set by the first icd command. P appears in conditional expressions with the execution bodies for the instructions. The last expression breaks the execution mode when BM detects some errors, and BM runs into the initialization mode. The C code of (1) represents a simple processor which decodes the comand icd, executes three instructions, waits for the command arrival in the while(Q) loop, and interupts the execution if error is detected.

2. Compilation

The input of the compilation stage are complex control-flow constructs including branches

and loops and data-flow parts. As we concern ourselves only with control generation, we suppose that the data-flow parts are synthesized through traditional algorithms[6]. The traditional approach (the reference stack algorithm in [6]) resolves constant and variable propagation, conditional assignment to variables, but it does not generate an FSM, but only a control-flow graph(CFG), where vertices represent assignment or conditional statements and edges represent the ordering of execution for each vertex. In our approach, the output of the compilation stage is a 2-level description binding data-flow components and control-flow constructs. The flattening process for the conditional nestings stops works as follows: (Cu, Mu(Bk's, (Cv, Mv)))=Mu(Cu&(Bk's), (Cu&Cv, Mv)), and if Cu is branch, $Mu=Mu_1$ (Cu&(Bk's)) + Mu_2((Cu&Cv, Mv)), where '+' denotes superposition of execution bodies. The data-flow constructs Bk's are broken down to simple data-flow expressions gk's having one assigned variable v_i as follows: $v_i=b_k$. These Bk's can be expressed as a sum of gk's, i.e., as a linear form. The compiler generates a vector space model LM (S,I,O) in a direct product space $V = V_M \times V_F$. V_M is a vector space over *GF(p)* spanned by a state vector S, an input vector I, and an output vector O. V_F is a Boolean space of conditional expressions. Let M be a vector whose components are referred by the variables on the left hand sides of the assignment operators "=", i.e., data-flow type variables in BM. For example, $(M)_s$ is a vector component referred by the variable 's': $(M)_s= M1_s+M2_s+M3_s= gs_1+gs_2+gs_3=$ (s=b+c)+(s=b*c)+(s=b*c+d*e). Let the conditional expressions Cu's be described with Boolean variables Yj's. All the nesting of control-flow constructs are flattened to a sum of products of Yj's at the compilation stage, and the products of Yj's are denoted by Boolean variables Xi's. Let F be a n-dimensional Boolean vector whose i-th component $(F)_i$ corresponds to the Boolean variable Xi. A complex expression, i.e., a group of expressions within the execution body selected by Xi, is denoted by Mi. The Xi control for simple data-flow expressions Mi is expressed by multiplying Xi and Mi. Thus, one can add the data-flow expressions to LM, and the compiler generates LM as a sum of products of M_i's and Xi's:

$$LM=M+F= \Pi\, Cu_i\&(\Sigma\, gk_j)+F= X0\&M0+X1\&M1+X2\&M2+...+Xn\&Mn+F(X1,X2,...,Xn), \quad (2)$$

where F is a controller which bind the Xi's to their conditional expressions so that there is no conflict among components. For example, the compiler binds $(F)_0$=P'&reset, $(F)_1$= ready&icd, and generates pairs of terms X0&(ready=1) and X0=P'&reset, X1&(s=b+c), X1=P&k, for M and F. The true-body of the 'if(k)' clause, k&(s=b+c), and the else-body, (k'&i)&(s=b*c) are added at the s-component of LM. The expression k&(s=b+c)+(k'&i) &(s=b*c) stands for $(M)_s$=k&(b+c)+(k'&i)&(b*c). Boolean variables do not have corresponding vector components, but vector components can refer to Boolean variables. For example, $(M)_{ready}$ can refer to the Boolean variable Y_{ready}, but the reverse mapping is

inhibited. Here we do not decompose nonlinear components, but we concern ourselves only with linear expression. The body of while(Q) of BM (1) can be compiled as follows:

$$M = (reset\&P')\&(ready=1)+(P\&k)\&(s=b+c)+(P\&k'\&i)\&(s=b*c)+(P\&j)\&(s=b*c+d*e)$$
$$+(ready\&icd)\&(P=true)+(error)\&(P=false) = X0\&g0+X1\&g1+X2\&g2+X3\&g3+X4\&g4+X5\&g5,$$
$$F=(X0,X1,X2,X3,X4,X5)=reset\&P'(F)_0+P\&k(F)_1+P\&k'\&i(F)_2+P\&j(F)_3+ready\&icd(F)_4+error(F)_5=1 \quad (3)$$

where gi's stand for primitive graphs of arithmetic or logical operators such as "+", "*", and "&", not" ' ", and the variable Q is omitted for simplicity. The templates of the gi's are supposed to be present in a library. The linear form of F is described with basic components of F, and (F)i's correspond to conditional vertices in the CFG. The next transformation steps consist of hardware allocation, module binding and scheduling, and control generation.

3. Derivation of loops from control-flow

Loops lp's are defined syntactically with conditions and loop bodies. For example, while(Q) has a condition 'Q' and a loop body. Semantically, a loop consists of a loop body and chains of in-coming edges defined with a condition. This definition is reasonable because a machine must stay in a finite set of states permanently if conditional variables Yi's do not change. The loops are utilized to make a finite state machine(FSM) in our approach. A system has a main (top) loop lp^0, and k-th level loops lp^k_i are appended to their upper loops lp^{k-1}_j recursively. Traditional high level synthesis generates similar state transition loops from the loops of a BM model by allocating controllers to the conditional vertices, but optimal loops for the state transitions are not obtained by a simple mapping from the loops of the BM model. An optimal loop of the FSM, which is a least *micro*-sequence of machine operations with smallest hardware components, may be different from the loops of the BM model which are *macro*-sequences to execute efficiently at the function level. Therefore we must reconstruct the loops of the BM model with *primitive* loops of micro-sequencies to get efficient micro-sequences. As some macro-equences are selected by subsets of Yi's, they are substituted with their micro-sequences under the same subsets of Yi's and other additional variables. The input of this stage is flattened codes and the output is a subset of products terms selected by Yi. Let us expand the logical expression lp^k by Shannon expansion with respect to the variable Yi which occurs more frequently in lp, then,

$$lp_a{}^k = Yi(Z_{a,i}{}^k + lp_b{}^k{}_{|Yi} + lp_c{}^k{}_{|Yi} + ..) + Yi'(Z_{a,i}{}^k + lp'_b{}^k{}_{|Yi'} + lp'_c{}^k{}_{|Yi'} + ..) = Z_{a,i}{}^k + Yi(lp_b{}^{k+1} + lp_c{}^{k+1} + ..)$$
$$+ Yi'(lp_b{}'^{k+1} + lp_c{}'^{k+1} + ..),$$

where $lp_b{}^k{}_{|Yi} = lp_b{}^{k+1}$, $lp'_b{}^k{}_{|Yi'} = lp'_b{}^{k+1}$, and $Z^k_{a,i}$'s do not include Yi and Yi' variables. This relation denotes that lp^k's are degenerated to $lp^{k+1}+Z^k_{a,i}$ when Yi=1 or to $lp'^{k+1}+Z^k_{a,i}$ for Yi=0. Then, the lp^{k+1}'s are expanded with respect to other variable Yj's recursively, and finally we get lp by composing the lpi's through Shannon expansion recursively.

$$lp^k=Z_i^k+Yi(Z_i^{k+1}+Yj(Z_j^{k+2}+Yq(Z_q^{k+3}+lp^{k+4}...)))+Yi'(Z'_i^{k+1}+...)))=$$
$$Z_i^k+YiZ_i^{k+1}+Y'iZ'_i^{k+1}+YiYjZ_j^{k+2}+..+YiYjlp^{k+3}+..+YiYj..Yrlp^{k+r}. \quad (4)$$

The equation (4) shows the *loop decomposition*. For example, a Shannon expansion of F with respect to variable P, that occurs more often in equation (4), is

$$F=reset\&P'(F)_0+P\&k(F)_1+P\&k'\&i(F)_2+P\&j(F)_3+ready\&icd(F)_4+error(F)_5$$
$$=P'\&reset'(F)_0+P\&(k(F)_1+k'\&i(F)_2+j(F)_3)+ready\&icd(F)_4+error(F)_5=P'\&lp1+P\&lp2+Z, \quad (5)$$

where F is binate in P. Both product terms are transformed into the loops lp1 and lp2. Those loops correspond to the primitive loop of the "while(Q)" and "if" statements in BM.

4. Hardware allocation

We define data flows by allocating register transfer level(RTL) templates. Our approach for the data-flow parts gi's is a traditional one, but we postpone allocating hardware for the control-parts to the logic synthesis step. Therefore we allocate no multiplexer but only function blocks and registers, and some data-paths among them are also determined by themselves. The data-flow parts gi's are replaced by templates of the RTL Gi's. For example, a binding of the data-flow parts gi's of equation (3) to Gi's is given by:

ready=true, P=true, P=false => G0(True, False, =, ready, P);
g1: s=b+c => G1(+, R1, Rd, Rd), 2xG2(=, D, R1), G3(=, R1, Rd), G7(=, Rd, s);
g2: s=b*c => G4(*, R1, Rd, Rd), 2xG2(=, D, R1), G3(=, R1, Rd), G7(=, Rd, s);
g3: s=b*c+d*e => 2xG4(*, R1, Rd, Rd), G5(+, R2, Rd, Rd), 4xG2(=, D, R1),
2xG3(=, R1, Rd), 2xG6(=, Rd, R2), G7(=, Rd, s);

where the Gi's stand for primitive graphs with registers Ri's, function blocks '+' and '*', and data paths'=' made with buses, multiplexers, or simple wire connections. The templates of Gi's are registered as elements of a linear space V_M, and they are described by matrices. The gk's and Gk's are represented by connection matrices Mgk's and MGk's whose entries gk(i, j) and Gk(i, j) denote edges from j-input to i-output, and function codes. The substitution of gi's with Gi's means the change of representation from behavior to data flow level, and the equation (3) can be written as

$$LM=X1\&(G1+2G2+G3+G7)+X2\&(G4+2G2+G3+G7)+X3\&(4G2+2G3+2G4+G5+2G6+G7)$$
$$+X0\&G00+X4\&G04+X5\&G05+P'\&lp1+P\&lp2+Z. \quad (6)$$

5. Step allocation and scheduling

Minimum numbers of cycles Sgi's(latencies) for Gi's are supposed to be available from a library. Overall latences are computed as usual in a bottom-up fashion, except for loops. Latencies Slp's for loops lp's are defined as a maximum among latencies Sxi's for data-flows gi's which are activated by conditional variables Xi's, i.e., $(F)_i$ in lp. For example, data-flow g0 is controlled by the loop lp1, and g1, g2 and g3 are controlled by lp2. Latencies Slp's are computed from the equation (6) and latencies Sgi's. In the equation (6) Gi's are replaced by (Gi, Sgi)'s with a one-to-one mapping.

$$LM= X1\&\{(G1,Sg_1)+2(G2,Sg_2)+(G3,Sg_3)+(G7,Sg_7)\}+X2\&\{(G4,Sg_4)+2(G2,Sg_2)+(G3,Sg_3)+(G7,Sg_7)\}$$
$$+X3\&\{4(G2,Sg_2)+2(G3,Sg_3)+2(G4,Sg_4)+(G5,Sg_5)+2(G6,Sg_6)+(G7,Sg_7)\}$$
$$+X0\&(G00,Sg_{00})+X4\&(G04,Sg_{04})+X5\&(G05,Sg_{05})+F(lp1,lp2,Z,P) \quad (7)$$

As V_M is a vector space, we can define a norm for the vector LM, and let define a norm with respect to latency S be a sum of Sg components as $|LM|s=\Sigma S_{gi}$. For example, norms $Sx_1=Sg_1+2Sg_2+Sg_3+Sg_7$, $Sx_2=Sg_4+2Sg_2+Sg_3+Sg_7$, and $Sx_3= Sg_1+4Sg_2+Sg_3+Sg_4+Sg_5+Sg_6+Sg_7$ are calculated, and a relation $Sx_1<Sx_2<Sx_3$ holds because the latency Sg_1 for "+" is less than the latency Sg_4 for "*" and all the Sgi's are possitive. A latency Slp2 equal to the largest latency Sx3 is required for the loop lp2. There are many ways to implement the control logic circuit for gi's with steps Sxi's, e.g. with hardware sequencers, adaptive control elements, or FSM's. To get the optimal FSM, which consists of simplest transition conditions with minimun number of state transitions, we must transform the control structures derived from the CFG to suitable control structures in a FSM. For example, we must merge three steps Sx_1, Sx_2 and Sx_3 to one step Slp2 with branches placed at some states.

6. Scheduling of loops , Loops in state transition graphs

A new scheduling technique of the loop lp's for a FSM model is proposed in this and the next sections. An FSM is defined by a state transition graph (STG) which consists of control inputs, a current state, and next state. Let us define LP to be *a loop body of a finite state machine* corresponding to the lp. The initial LP is given simply by dropping Gi's from the equation (7):

$$LP=P'\&X0\&Sg_{00}+P\&\{X1\&(Sg_1+2Sg_2+Sg_3+Sg_7)+X2(Sg_4+2Sg_2+Sg_3+Sg_7)+X3\&(4Sg_2+2Sg_3+2Sg_4$$
$$+Sg_5+2Sg_6+Sg_7)\}+X4\&Sg_{04}+X5\&Sg_{05}+F(lp1,lp2,Z,P)=P'\&LP1+P\&LP2+Z+F, \quad (8)$$

where P' and P indicate explicitly control of the LP's. Thus, by using loop body decomposition, we can divide LP into smaller loops LPi's. The Sgi's in LP2 of equation (7) are arranged in execution order with respect to data dependencies as seen in usual procedures. Terms are rearranged by factorizing out the Xi's.

$$Lp2=X1(Sg_2+Sg_3+Sg_2+Sg_1+Sg_7)+X2(Sg_2+Sg_3+Sg_2+Sg_4+Sg_7)+X3(Sg_2+Sg_3+Sg_2+Sg_6$$
$$+Sg_2+Sg_4+Sg_3+Sg_6+Sg_2+Sg_4+Sg_5+Sg_7) = (X1!X2!X3)(Sg_2+Sg_3+Sg_6+Sg_2+Sg_7)$$
$$+X3(Sg_2+Sg_4+Sg_3+Sg_6+Sg_2)+X1Sg_1+(X2!X3)Sg_4+X3Sg_5 \quad (9)$$

where symbols "!" mean 'or' operators, and mSgi's are divided separately because of the linear condition: (m1+m2)Sg=m1Sg+m2Sg. A similar equation for the Gi's holds by replacing Sgi's with Gi's. The rearrangement means hardware sharing in loop lp2 Hardware sharing in loop lp1 and in Z is done separately. Though controls are separated into loops, each loop controls global data-flows. Therefore hardware sharing over global data-flows is obtained, and global optimizations are achieved. Each Boolean equation is optimized separately.

7. State allocation and state transition control

To get an optimal FSM after rearrangement of Sgi's in execution orders, traditional resource sharing is taken into consideration in many scheduling procedures, for example, in the *As soon As Possible* or *As Late As Possible* algorithms and so on. The equation (9) is divided into time slices by considering registers allocation and state cycle time as in traditional scheduling algorithms:

$$Lp2=(X1!X2!X3)(Sg_2+Sg_3+Sg_2)+X3(Sg_2+Sg_4+Sg_3+Sg_6+Sg_2)+X1Sg_1+(X2!X3)Sg_4+X3Sg_5+(X1!X2!X3)Sg_7 = (X1!X2!X3)([Sg_2]_{S6}+[Sg_3+Sg_2]_{S2})+X3([Sg_6]_{S2}'+[Sg_2+Sg_4]_{S5}+[Sg_3+Sg_6+Sg_2]_{S2''})+[X1Sg_1+(X2!X3)Sg_4]_{S3}+[(X3Sg_5+(X1!X2!X3)Sg_7)]_{S7} \quad (10)$$

where S6, S2, S2', S2", S5, S3, and S7 are state labels (see next section's convention). In our approach, one gererates the state transition chains S6>S2>S3>S7 for X1 and X2, and S6>S2>S2'>S5>S2">S3>S7 for X3 by tracing each Xi. The states S2, S2' and S" are merged by traditional merging and folding algorithms to get the new state sequence S6>S2>S5>S2>S3>S7, and finally the loop S6>S2>(S5>S2 for X3)>S3>S7>S6 is deduced by merging same states. This causes hardware sharing. For example, G2, G3, and G6 are shared at S2. Then state transition control variable I (1bit for this example) is generated to control the branch at S2 from Xi's and P, and Xi's control steps Sgi's in each state. Thus, an FSM is generated by combing I and Lp2. Similar equations for the other loop Lp1 and Z are obtained: $Lp1=X0[Sg_0]_{S0}$, $X4[Sg_4]_{S4}$, $X5[Sg_5]_{S5}$. The loops are indexed by sets of the state transition control variables(see section 2.1). Each state is ordered by weights of resources. Also, the loops are ordered by the weights of states for use of optimization of LM. The state transitions from one loop to another are added, and the control variables are generated for newly specified transition conditions at the upper loop level. The loops are divided into two types of global and local types by their corresponding global or local data-flow parts. The global type loops specify global control variable Xi's which one can use for global optimization of data-flows. ***linearity*** of loops insures no conflict with existing components X_i&M_i's when additional components X_{n+1}&M_{n+1} are appended to it. This definition is different from the one of linear machines, but it still reflect the principle of superposition.

2.1 FSM Example

Consider a simple example data-flow level model FSM1. It was first presented in [8] and the corresponding state table is described in **Table 1**. The equation (10) is transformed to it, assuming 1 data input port D(data), 3 registers, and one "+" or "*" for each state.

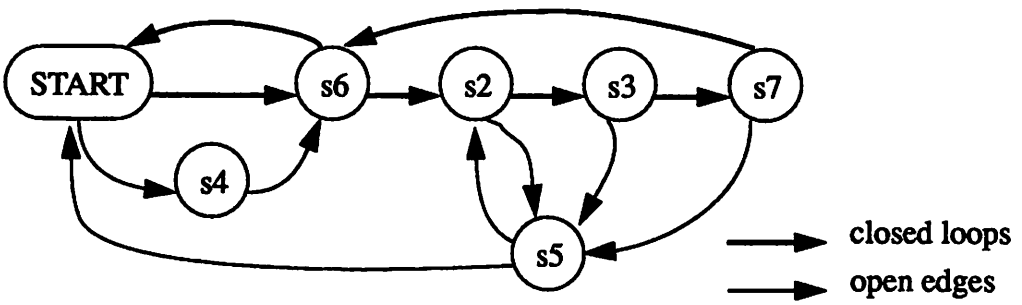
```
    k=(icode==add); i=(icode==mul); j=(icode==sop); m=k'&i; n=k'&i'&j;
START: k=0; i=0; j=0; if(I) next S4; else next S6;
S4: Ri = 0; Ready=1; next S6;
```

S6: R1 = D(b); if(I) next S2; else next START; // G2
S2: Rd = R1; R2 = Rd; R1 = D(c ! e); if(I) next S3; else next S5; // G3+G6+G2
S5: Rd = Rd*R1; R1 = D(d); if(I) next S2; else START; // n(G4+G2)
S3: Rd = k(Rd+R1)+(m!n)(Rd*R1); if (I) next S7; else next S5; //kG1+(m!n)G4
S7: s = n'Rd+n(Rd+R2); if (I)next S6; else S5; // G7+nG5

where "!" stands for OR operators, "&" for AND operator, registers are "Ri", state identifiers are "Si", and next state designators "next". The thin font "+" operators are used for multiplexers or busses. The state transition matrix A is derived from the state transition control parts of FSM1. The STG has four loops: 1) Lp1: START <-> s6 for condition (I=0); 2) Lp2: s6> s2> s3> s7> s6 for condition (I=1); 3) Lp3: s2> s5> s2; and 4) Lp4: START> s4> s6> START.

Table 1: State Transition of FSM1 example

Current state	Next state I=0	Next state I=1	Output I=0	Output I=1
START	state-6	state-4	00	01
state-2	state-5	state-3	00	10
state-3	state-5	state-7	00	10
state-4	state-6	state-6	00	10
state-5	START	state-2	10	00
state-6	START	state-2	01	00
state-7	state-5	state-6	00	00



The 7x7 dimensional matrix A is divided into two parts: the loops and chains of the edges. The sub matrices of A for the loops Lp1, Lp2, Lp3 and Lp4 are described with non zero entries xi's expressing state transitions, and edges: $E_{3,5}$= (s3,s5), $E_{7,5}$= (s7,s5), $E_{5,st}$ = (s5,START) where $E_{si,\ sj}$ have one non zero entry in the s_i-row and the s_j-column. A complex loop Lp5 is obtained by binding them linearly,

$$Lp5 = X1\begin{bmatrix} 0 & x1 \\ x2 & 0 \end{bmatrix}\begin{pmatrix} s6 \\ ST \end{pmatrix} + X2\begin{bmatrix} 0 & x3 & 0 & 0 \\ 0 & 0 & x4 & 0 \\ 0 & 0 & 0 & x5 \\ x6 & 0 & 0 & 0 \end{bmatrix}\begin{pmatrix} s7 \\ s3 \\ s2 \\ s6 \end{pmatrix} + X3\begin{bmatrix} 0 & x7 & 0 \\ 0 & 0 & x8 \\ x9 & 0 & 0 \end{bmatrix}\begin{pmatrix} s6 \\ s4 \\ ST \end{pmatrix} + X4\begin{bmatrix} 0 & xa \\ xb & 0 \end{bmatrix}\begin{pmatrix} s5 \\ s2 \end{pmatrix} \quad (11)$$

where coefficients Xi's are products of complex coefficients (X_{Ci}) and logical coefficient

(X_{Li}) respectively, and the sub matrices consist of rows including only non zero entries xi's for simplicity. Each row of the sub matrices corresponds to the same next state described in the same row of the column vectors on their right side. The same state will be written in different forms within different loops Lpis, but they have to be self consistent so that we can thread the loops in a chain. If Lpi has no change of condition X_{Li} through all the state transitions in Lpi, the loop Lpi is said a closed n-state loop with condition X_{Li} or a (n:X_{Li})-loop in short. The Table 1 shows that Lp1 with X_{L1}(I=0) and Lp2 with X_{L2}(I=1) are closed loops. The closed loops are distinguished by unique control conditions so that we regard them as indexed loops. Lp3 and Lp4 are not closed since $X_{L3}=X_{L30}$(I=0| s4,s6) +X_{L31}(I=1| s4,ST) and $X_{L4}=X_{L40}$(I=0| s2)+X_{L41}(I=1| s5). State s4 with edges (START,s4) and (s4,s6) belongs to Lp1, and the s5 with the edges of (s2,s5), (s3,s5),(s7,s5) and (s5,s2) does to Lp2. The two other edges (s6,s6) and (s5,START) are paths from/to Lp1 and Lp2. The states at which the out-edges are attached, are said gated states, and the loops with m gated states are said m-gated loops or m-gated open loops. We note that the loops are irreducible components which cannot be represented in linear form.

2.2 Basic Structure of Algorithm

The basic structure of algorithm which we are developing, is summarized here:

1. Find Loops in flow graph: classify the control flow graph(CFG) with respect to the transition conditions. Build matrix M(i,k) with matrices Mi(i,k) for each data-flow graph (DFG) and F(i,k) for conditions Ci(i,k) and calculate resource weight.
2. Find the initial loop Lp(i,k,0) in F(i,k) and set the loop level=0. Then find next loops Lp(i,k,l+1) by Shannon expansion with respect to the more often occuring conditional variables in the current loop Lp(i,k,l) recursively. Compute the number of branches B#(i), and specify the state control input I(n#). Calculate the number of states in each loop S#(i), and specify state code width C#(i). Put weight W(i,j,k). Merge and reduce the loops to the diagonal elements of M(i,j,k) by reducing related rows and columns.
3. Find bridges among lower loops. Check merge conditions among loops. Delete a loop if it is merged into another loop. Merge C#(i), I(n#)
4. Select the first loop in which the system starts, and also select the initial state. Allocate an initial state code with a code width C#(i), and calculate the state codes of a subfield successively for all the states from the initial state. Find reachable loops, and find states connected with bridge transitions. Allocate the most suitable code to them. Then allocate codes to the successive states recursively.
5. Put the most suitable codes on the other states out of loops. Hand over to a state assignment programs with state codes constraints for generating state transition circuits.

2.3 State coding experiments with Johnson Counters

The optimal *random* coding results by a state assignment program NOVA for the symbolic cover FSM1 example is obtained as CASE-I [(s0 s6 s4 s2 s5 s3 s7)=(110, 000, 111, 010, 100, 001, 011)] in Table-II, where outputs=(O0, O1), an input.=(I), latch-inputs=(LIi's), and latch-outputs=(Li's). Let us examine the effects of embedding counters into the system. As an Nbit-Johnson counter can express 2N states, FSM1 needs 4 Johnson counters at most. If all loops do not intersect each other, they have 1-bit, 2-bit, 2-bit and 1-bit code width. We note that sequences of state codes are specified uniquely except an arbitrary code for the initial state. If we choose initial state codes to be 0, state codes of counters are defined as follow: Lp1:{0,1}, Lp2:{00,10,11,01}, Lp3:{00,10,11}, Lp4:{0,1}. If we take simply a direct products of all the coding fields, then we have 6 bits width coding equal to 1+2+2+1. It can satisfy the constraints of intersections between Lp1 and Lp2 at s6, Lp1 and Lp3 at START and s6, or Lp2 and Lp4 at s5. The obtained coding is

Lp1: {START,s6}={0--00-, 10011-}; Lp2:{s6, s2, s3, s7}={10011-, -10--0, -11---, -01---};
Lp3:{START, s4, s6}={0--00-, ---10-, 10011-}; Lp4:{s2,s5}={-10--0, -----1};

The don't-cares (-)'s in the codes are fixed one after another by satisfying the constraints under which each code can change only one bit value at one transition along any loop for a racing free condition. The obtained result is

Lp1: {START,s6}={000000, 100110}; Lp2:{s6, s2, s3, s7}={100110, 110110, 111110, 101110};
Lp3:{START, s4, s6}={000000, 000100, 100110}; Lp4:{s2,s5}={110110, 110111};

The bit width is twice longer than the minimum one. FSM1 runs from one loop to another loop under one bit flip at once. This property is necessary for hazardless controllers. NOVA outputs the Case-II circuit under the above state codes constraints. It shows that the latch L0 is redundant because 100000 is an external don't care. A Johnson counter for the loop Lp2 is automatically allocated with L1 and L2 latches. The longest chain under the condition I=1 is the closed loop Lp2 with in-edge {START, s4, s6} and it produces the smallest circuit with a 3bit Johnson counter {START,s4,s6,s2,s3,s7}={000,100,110,111,011,001} and s5={010}. The Case-III, with a Johnson counter, saves 4 product terms from the random coding Case-I. Next we consider the second coding Case-IV with the largest loop Lp1':{START, s4, s6, s2, s3, s7, s5}={0000, 1000, 1100, 1110, 1111, 0111, 0011} and Lp2. The system has a less efficient implementation with no Johnson counters, and also there are 7 violations at the state transitions {START, s6}, {s6, START}, {s7, s6}, {s5, START}, {s3, s5}, {s2, s5}, {s5, s2}. The reason is that the loop is not a closed loop, and that the transition to the additional state s5 is inhibited under I=1. These examples imply that we must choose the largest closed loop for embedding a counter as we expected. The Case-V is based on the fact that Lp3 intersects Lp1 in the states ST and s6, and so we can drop one bit for Lp1 to get a 5bit Johnson counter with the state codes

Lp1: {START,s6}={00000, 01000}, Lp2:{s6, s2, s3, s7}={01000, 01100, 01110, 01010},
Lp3: {START,s4,s6}={00000, 10000, 01000}, Lp4:{s2,s5}={01100, 01101}.

We note that the simplest circuit is obtained without feedbacks from the outputs. If we delete open loops Lp3 and Lp4, we have the 3bit Johnson counter of Lp1:{START,s6}= {000,100}, and Lp2:{s6, s2, s3, s7}={100,110,111,101}. The other states (s4, s5) can be selected from the remaining codes (011,001,010). These are 6 possible cases:

(s4,s5|sop)=(001,010| 35),(001,011| 37),(010,001| 33),(010,011| 38),(011,010| 33),(011,010| 30)

The simplest circuit is obtained with the code (011,010| 30), but it has feedbacks from the outputs.

Table 2: Synthesized results with state code assignments

Coding	Nodes	Latches	Sops	Stages
I	7	3	32	7
II	7	5	30	7
III	5	3	28	7
IV	6	4	38	7
V	7	5	29	7
VI	5	3	30	7

------ CASE I ----------

O0 = I L3 [21]+I L3'[21]'+I' L1 L2' ; O1 = O1'LI1 LI2 ; LI1 = I' L1' + I' L2' +L1[19] ;
LI2 = I L2' + L1 [19] + L2' L3' ; LI3 = L3 LI2 + [19] ; [19] = I L2 L3' ; [21] = L1 +L2'

-----CASE-II ----------

O0 = I L3 L4' + I'L5 +IL2; O1 = I L3' + I' L1' L2' L4 ; LI1 = I L2' L4 + LI5 ;
LI2 = I L1 L5' ; LI3 = I + LI4 ; LI4 = O1' L4' + L2 + LI1 ; LI5 = I L2 L5' + I' L2

-------CASE III ----------

O0=I L1 LI3' + L1' L2 L3'LI3' + L3 LI3 : O1=I L1' L3' LI3' + L1 L2 L3' LI3' ;
LI1=O0' O1'L3' +I LI3' ; LI2= O0' O1' + O1' L1 ; LI3=I L2 ;

-------CASE IV ---- ----------

O0=I L1 LI2' + I LI4 + L2' LI1' : O1=I L1' L4' + I' L2 L3' ;
LI1= I L1' +I L2' + I L4' +L2'L4' ; LI2= O1' LI1 + I LI3 ; LI3= L2 L4' LI1 + L2' L4 LI1 +LI4 ;
LI4 = I L1 L3 + I' L2 L3

--------V----------

O0=I L3 L5' + I L1 + I' L5 ; O1=I L1' L2' + I' L2 L3' L4' ; LI1=I L1' L2' ;
LI2=(I L1' L2')'L2' + L4 + LI3 ; LI3= I L2 L4' + LI4 ; LI4=I L3 L5' ; LI5 = I' L3 L5' + I' L4;

--------VI--------- ----------

O0=I L2 L3 + L2 L3' LI2' + L2 LI3 : O1=I' L1 L2' L3' + L2' LI3';
LI1=O0' L1' LI2' + O1' I ; LI2=O1' I' L1 + I L3' ; LI3=I L1 L2 + I L1' L2';

3.0 Conclusions

This paper presents a framework for the high-level synthesis of FSM's from the control-flow graphs and it links the domains of Linear systems and digital systems. FSM's are synthesized from control and data flow graphs by combining paths of state transitions, while considering resource costs for the data paths. A mathematical base for analyzing high level

design is proposed. Linear group theory is effective to get symmetry or regularity in the system (the closed loops are defined to be invariant under the index condition). There is a lot of work yet to do to get more efficient algorithms. We note that primitive loops must be classified mathematically for the purpose of analytical approach in future.

REFERENCES

[1] D.C.Ku, G. De Micheli,"High level Synthesis of ASICs Under Timing and Synchronization Constraints", Kluwer Academic Publishers 1992

[2] R.E.Bryant,"Graph-Based Algorithms for Boolean Function Manipulation",IEEE Trans on Computers, vol.C-35 No.8 August 1986

[3] R.Amann and U. G. Baitinger,"Optimal State Chains and State Codes in Finite State Machines", IEEE Trans. on Computer-Aided Design, pp153-170, Vol. 8 No. 2 Feb. 1989

[4] L.Stok,"False Loops through Resource Sharing", Proc. of ICCAD'92, pp345-348, Nov. 1992

[5] M.Nourani and C.Papachristou,"Move Frame Scheduling and Mixed Scheduling-Allocation for the Automated Synthesis of Digital Systems", Proc. of DA conf., pp99-105, June 1992

[6] M.Potkonjak, J.Rabaey,"Maximally Fast and Arbitrarily Fast Implementation of Linear Computations", Proc. of ICCAD'92, pp304-308, Nov. 1992

[7] C. Duff and G.Saucier, "State Assignment Based on the Reduced Dependency Theory and Recent Experimental Results", Proc. of ICCAD'91, pp222-225, Nov. 1991

[8] G.De Micheli, R.K.Brayton,"Optimal State Assignment for Finite State Machines",IEEE Trans. on CAD, Vol. CAD-4. No.3, pp.349-365, July 1985

[9] K.Wakabayashi, H.Tanaka, "Global Scheduling Independent of Control Dependencies Based Condition Vectors", 29th DAC, 1992

[10] T.Villa, A.Sangiovanni-Vincentelli, "NOVA:State Assignment of Finite State Machines for Optimal Two-Level Logic Implementation", IEEE Trans. CAD, 1990

[11] S.Devadas, H. Ma, R.Newton, A.Sangiovanni-Vincentelli, "Mustang: State Assignment of Finite State Machines Targeting Multilevel Logic Implementations", IEEE Trans. CAD, 1988