CSG: Control Path Synthesis in the ADAM System

Jen-Pin Weng and Alice Parker

CENG Technical Report 92-03

Department of Electrical Engineering – Systems
University of Southern California
Los Angeles, California 90089-2562
(213)740-4476

CSG : Control Path Synthesis in the ADAM System

Abstract

CSG (Control Signal Generator) automatically synthesizes controllers for both pipelined and non-pipelined data path designs with conditional branches which are produced by high-level synthesis tools in the ADAM system. The ability to synthesize controllers for pipelined designs with conditional branches contributes to the problem complexity. *CSG* has the ability to synthesize controllers with or without status registers. The purpose of introducing status registers into a controller design is to store the condition state in a design with conditional branches. Because of partially-specified values in the status registers, a multiple-code state encoding may be obtained in a controller using status registers. Using a multiple-code state encoding instead of a unicode state encoding enables *Espresso* to produce a better minimization. We experimented with nine non-pipelined designs and fourteen pipelined designs using a robot arm controller example. The experimental results are quite encouraging, especially in the experiments involving non-pipelined designs.

This work was supported by the Semiconductor Research Corporation under contract 92-DJ-075.

1 Introduction

Many design problems of practical interest exhibit conditional branching behavior. However, pipelined designs which contained conditional branches were not given serious thought by high-level synthesis researchers until the mid '80's, when such pipelined data paths were first synthesized [9]. The control of such data paths was only recently addressed [5].

The synthesis program described in this paper generates a controller specification from a description of system behavior and the respective data path logic. The purpose of the control unit is to issue control signals to the synthesized data path logic. These control signals select the operations to be performed at specific time steps and route the data through the appropriate functional units. Here, we are particularly interested in automatically synthesizing controllers for both pipelined and non-pipelined data path designs produced by high-level synthesis tools such as *Sehwa, MAHA* and *MABAL* with and without conditional branches.

The control path synthesis program CSG described in this report is part of the ADAM system. The DDS [6], the internal representation in the ADAM synthesis subsystem, maintains separate representations of data flow and control/timing information. Data path scheduling, module allocation and module binding are assumed to have been performed before control path synthesis to provide precise information on timing and values of all data path control signals. This information is incorporated into the control/timing behavior and condition values[1] to obtain correct controller specifications.

To retain flexibility in choosing the controller implementation style, only the behavior of the finite state machine is produced by CSG. Flexibility is important because we wish to synthesize digital circuits that have proper functionality and also meet the performance and area constraints. To accomplish this, the controller implementation style may vary from problem to problem. In this paper, we will focus on controllers characterized by a state transition diagram of a finite state machine. The controller is assumed to be implemented by a PLA circuit. However, if a microprogrammed implementation is preferred, the finite state machine can also be implemented with a microprogrammed controller.

[1]Condition values are values which affect system control flow by controlling branches in the execution. A condition value can be either input externally or created by the data path. In a two-way branch, the condition value typically takes the value 0 or 1 (*false* or *true*).

Due to the limited length of this paper, we will briefly introduce algorithms we used in our approach. A complete report of controller implementation procedures and experimental results will be found in [13]. In the next section, we present the related research work first. We then introduce the assumptions made in our approach and the controller implementation algorithms. Following the presentation of experiments performed by our system, conclusions are given.

2 Related Research

In the area of automated controller synthesis, most existing high-level synthesis systems only synthesize controllers for *non-pipelined* designs. A widely applied controller style in many existing synthesis systems is the ROM-based microprogrammed controller model. Examples include the control allocator for the CMU-DA system [8] and the ATOMICS system for Cathedral-II [4] [2]. The control allocator for the CMU-DA system assumes a canonical microprogrammed model, and performs optimizations based on the microcode format constraints. The ATOMICS system takes an RT-level description as input, and performs microprogram scheduling in order to minimizes the global machine cycle count.

Some systems map the control/data flow graph representing the hardware behavior into a corresponding state transition diagram, which can be implemented by either a PLA or random logic. For example, the Yorktown Silicon compiler [3] implements the controller as a hierarchy of finite state machines, where an FSM is associated with each routine. Control state splitting allows the delay through the combinational part of the data path to be reduced to satisfy clock cycle constraints. AT&T's Bridge system [11] [12] first creates a symbolic control table once the data path is allocated. The symbolic control table basically identifies all the modules which need to be activated in each cycle. The detailed implementation of the controller will be decided by the module generator. In the Hyper synthesis system [1], a state transition diagram is derived from a control/data flow graph based on the scheduling information. It is a recursive procedure due to the hierarchical nature of the control/data flow graph. The transition diagram is then optimized by removing the dummy states.

Kim's controller synthesis research at UC-Irvine [5] is the only reported work we know of which synthesizes controllers for pipelined designs. The controller is modeled as a Moore-style finite state machine. The control states are determined by the possible execution modes in a scheduled control/data flow graph using a coloring scheme described by Park [9]. The state transitions for each control state are then determined by the compatibility of two possible execution modes between two consequent groups of states. A condition value needs to be produced at least two clock cycles before the conditional execution is performed, because of the Moore-style controller.

3 Problem Assumptions

We assume that the register-transfer-level structure is divided along functional boundaries into two parts — the data path and the control path. The data path consists of a network of functional units, registers, multiplexers and buses. The control path generates the control signals required by registers, multiplexers and buses for sequencing of events in the data path, as well as the signals required by the control path itself[2]. In synchronous systems, the only kind we consider in this paper, a controller can be specified by a finite state machine which can be implemented by PLA profiles, random logic or microprograms.

Memory elements in the synthesized circuits are classified into three categories — *state registers*, *output registers* and *status registers*. State registers are used to keep track of the current state and partially decide the next state. Output registers are designed to hold the control signals to avoid possible race conditions between the data path and control path during the data path execution. Status registers are used to maintain the condition values, if the status-register controller implementation style is assumed.

A conservative two-phase non-overlapped clocking scheme has been assumed in our approach. A data path clock, $Clock_{DP}$, is always followed by a control path clock, $Clock_{CP}$. Positive edge-triggered D-type flip flops are proposed to implement all the memory elements, in both data path and control path, in this clocking scheme. All the values in the registers of a controller can only be changed at the rising edges of control-path clocks, and all the values generated by the data path can be stored into data-path registers only at the rising edges

[2]ALU control signals are not supported in the current CSG. However, they are similar to control signals for multiplexers and can be handled with a simple extension of the current CSG.

of data-path clocks. If each clock has a sufficiently long period then the correct execution of both control path and data path can be ensured and the possibilities of race conditions can be avoided. Note that the clocking scheme we defined above implicitly requires the condition values to be created at least one clock cycle before they can be used to determine which conditional execution paths are executed.

The controller is modeled as a Mealy finite state machine[3]. The state memory of the controller holds the present state, and a combinational circuit decides the next state and output signals. Once a condition value is produced, the controller will maintain it until no more operation executions depend on it. This approach simplifies the register allocation and binding subtasks of data path synthesis. Depending on the selected implementation style, status registers may or may not be used in the controller. The purpose of introducing status registers into a controller design is to store the condition state in a design with conditional branches. If status registers are not allowed in a controller design, the condition state will be stored in the state memory, which implies more state memory is required.

Because of partially-specified values in the status registers, an efficient multiple-code state encoding may be obtained in a controller using status registers. Using a multiple-code state encoding instead of a unicode state encoding usually enables *Espresso* to produce a better minimization. We experimented with 9 non-pipelined designs and 14 pipelined designs using the robot arm controller example. The experimental results show that 15% to 20% less area is required for the PLA/controller on the average when status registers are used, especially in the experiments involving non-pipelined designs.

4 Implementation of Controllers

Three parameters need to be fixed when designing a controller, namely *the number of states needed for the controller*, *the next-state transitions for the current state* and *the control signals output for each state.* In this section, we start with the determination of the possible control signals. The implementation of controllers for designs without conditional branches is then given. We discuss the implementation of controllers for designs with conditional

[3]The outputs of a Mealy finite state machine depend on inputs as well as the current state. The outputs of a Moore finite state machine depend on the current state only.

branches last. Controllers for both pipelined and non-pipelined designs are discussed.

4.1 Control Signal Generation

Before a controller can be realized, we first need to determine all the possible control signals which are required during the execution. Control signals are represented as tuples. A *tuple* has several attributes, such as *activated device name, activated time step* and *activating condition values*[4]. For a multiplexer control signal, an extra attribute — *activated port number* — is used.

In order to determine the condition value requirements for the execution of operations along the conditional branches, an analysis of the control/data flow graph mutually exclusive conditions is necessary. We analyze the mutually exclusive conditions of a control/data flow graph using an algorithm similar to the one proposed by Park [9]. The algorithm assigns to every node a color consisting of a sequence of one or more integer codes. Using these colors, we can test the possible mutual exclusion between any pair of nodes (operations) and obtain the condition values for activating the operations.

Two types of binding information are used in our approach. An *operation binding* binds an operation to an operator, and a *value binding* binds a data value to a register. For an operation binding, the input data values of this operation are expected to come from external inputs or register outputs; the input of a register should come from external inputs or an operation's output. Three types of control signals may be produced, namely *register write signals*, *multiplexer port select signals* and *tri-state-driver enable signals*. Those control signals are obtained by combining the binding information and the respective register-transfer level network. A set of control signals is issued by a controller to achieve the activities specified in the binding information under the proper timing sequence.

4.2 Controllers Without Conditional Branches

The controller specification in this class of designs is simple and its state transition diagram is a ring. The time steps are a continuous sequence of numbers for a scheduled control/data

[4]The activating condition values are a set of condition values which is required by the execution of an operation.

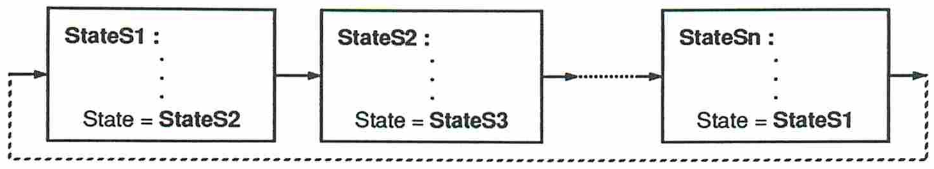

Figure 1: A Ring State Transition Diagram

flow graph. In our approach, the vertices of a scheduled n-time-step control/data flow graph, $\mathcal{G}_n$, can be clustered into a set of *executions* $\mathcal{E} = \{\mathcal{E}_1, \mathcal{E}_2, \ldots, \mathcal{E}_n\}$. An *execution* $\mathcal{E}_i$ is then defined as a set of the control/data flow graph operations which are scheduled to be executed in time step i for $i = 1 \ldots n$. Note that, the set of vertices of a control/data flow graph is the union of the execution set and set of distribute/join nodes[5]; and the *executions*, $\mathcal{E}_i$ and $\mathcal{E}_j$, are disjoint for $i, j = 1 \ldots n, i \neq j$. Thus, the task of a controller is to generate a set of control signals to ensure that all the operations in $\mathcal{E}_i$ are to be executed according to specified behavior in time step i for $i = 1 \ldots n$.

A state transition diagram $\mathcal{T}_n$ is a graph with the *states* $\mathcal{S} = \{\mathcal{S}_1, \mathcal{S}_2, \ldots, \mathcal{S}_n\}$ as its vertices and the *transition arcs* as its edges. A *state* consists of a set of control signals which will be activated when the state is being visited. A *ring state transition diagram*, $\mathcal{R}_n$, is a state transition diagram, $\mathcal{T}_n$,whose vertex set is $V = \{\mathcal{S}_1, \mathcal{S}_2, \ldots, \mathcal{S}_n\}$ and whose edges are $\{\mathcal{S}_1, \mathcal{S}_2\}, \{\mathcal{S}_2, \mathcal{S}_3\}, \ldots, \{\mathcal{S}_{n-1}, \mathcal{S}_n\}, \{\mathcal{S}_n, \mathcal{S}_1\}$. A ring state transition diagram, $\mathcal{R}_n$ is necessary and sufficient for a non-pipelined design without conditional branches.

The controller is realized by defining a *bijective* function, $\mathcal{F} : \mathcal{E}_i \rightarrow \mathcal{S}_i$ for $i = 1 \ldots n$. The *control signal generating function* $\mathcal{F}$ produces a set of control signals which are activated during execution of operations within the *execution* $\mathcal{E}_i$; and the set of control signals is assigned to *state* $\mathcal{S}_i$. By assigning the first state to state $\mathcal{S}_1$, and so on the controller specification for a non-pipelined design is completed.

Designing the controller for a pipelined design is similar to the procedure for a non-pipelined design. The basic idea is to "*fold*" the scheduled control/data flow graph every initiation-interval[6]. The number of total time steps of this new control/data flow graph is equal to the length of the initiation interval; and each operation in the control/data flow graph will be executed every initiation interval time step. The formal model of a folded

[5]The distribute/join nodes in a control/data flow graph are not assumed to be assigned to any time step in the current ADAM system.

[6]An initiation interval is the number of time steps between introductions of new data into a pipeline.

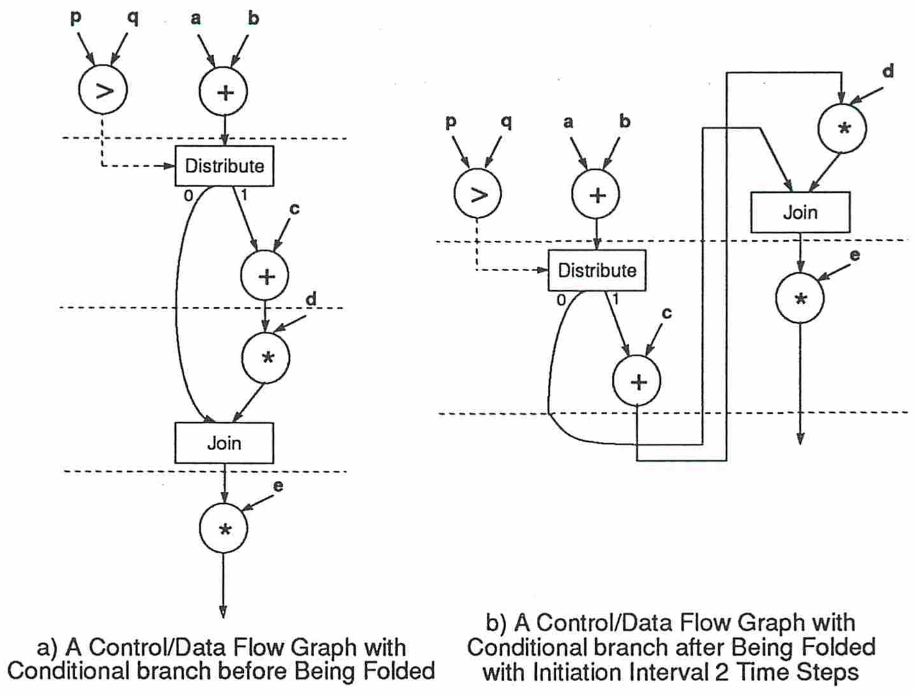
p
q
a
b
>
+
Distribute
0
1
c
+
d
*
Join
e
*
a) A Control/Data Flow Graph with Conditional branch before Being Folded
b) A Control/Data Flow Graph with Conditional branch after Being Folded with Initiation Interval 2 Time Steps

Figure 2: A Folded Control/Data Flow Graph

control/data flow graph for the pipelined design is given as follows. The vertices of a *folded* scheduled n-time-step control/data flow graph, $\mathcal{G}_{pn}$, for a pipelined design with initiation interval I can be represent by a set of *p-executions*, $\mathcal{E}_p = \{\mathcal{E}_{p1}, \mathcal{E}_{p2}, \ldots, \mathcal{E}_{pI}\}$. A *p-execution* $\mathcal{E}_{pi}$ is defined as the union of executions $\{\mathcal{E}_i, \mathcal{E}_{i+I}, \mathcal{E}_{i+2I}, \ldots, \mathcal{E}_{i+mI}\}$ for all $i + mI \leq n$. The *p-executions*, $\mathcal{E}_{pu}$ and $\mathcal{E}_{pv}$, are disjoint for $u, v = 1 \ldots I, u \neq v$. A ring state transition diagram, $\mathcal{R}_I$, is necessary and sufficient in this case.

To realize this type of controller for a pipelined design, we also define a *bijective* function, $\mathcal{F}_p : \mathcal{E}_{pi} \rightarrow \mathcal{S}_i$ for $i = 1 \ldots I$. The *control signal generating function* $\mathcal{F}_p$ produces a set of control signals associated with *p-execution* $\mathcal{E}_{pi}$ which is the union of the control signals activated in time steps $i, i+I, i+2I, \ldots, i+mI$ for $i + mI \leq n$. The set of control signals is then assigned to the state $\mathcal{S}_i$. By assigning the state transition diagram to be started from the state $\mathcal{S}_1$, the controller specification for the pipelined design is done.

4.3 Controllers Using Status Registers

For a design with conditional branches, either each conditional branch is represented by a separate set of states or the condition values are "remembered" by the controller until no more conditional execution paths are dependent on those condition values. A controller which uses separate status registers to keep track of condition values is now presented. In general, a controller using status registers has a simpler state transition diagram than a controller without status registers. Our experimental results show the status-register implementation style also provides a potential way to reduce controller area cost.

The first time step a condition value, $\mathcal{V}_c$, is created and stored into a register is defined as $\mathcal{B}_v$ where $\mathcal{B}_v \geq 1$, and $\mathcal{V}_c$ is said to be *born* in time step $\mathcal{B}_v$. The last time step that the executions of operations are dependent on the condition value $\mathcal{V}_c$ is defined as $\mathcal{D}_v$ where $\mathcal{D}_v \geq \mathcal{B}_v$, and $\mathcal{V}_c$ is said to be *dead* in time step $\mathcal{D}_v$. The *lifetime* of the condition value $\mathcal{V}_c$ is thus defined as a span from time step $\mathcal{B}_v + 1$ to time step $\mathcal{D}_v$. The condition value $\mathcal{V}_c$ is said to be *alive* within the lifetime. The *reserved period* of the condition value $\mathcal{V}_c$ is defined as a span from time step $\mathcal{B}_v + 2$ to time step $\mathcal{D}_v$ if $(\mathcal{B}_v + 1) < \mathcal{D}_v$. Otherwise, the reserved period of the condition value $\mathcal{V}_c$ is defined to be `NIL`. The condition value $\mathcal{V}_c$ is said to be *reserved* within the reserved period.

The death of a condition value is not only decided by the schedule but also by the module bindings and allocations. The death of condition values can be obtained from *condition-value lifetime analysis.* The condition-value lifetime analysis inspects all the control signal tuples; and the death of condition value $\mathcal{V}_c$ is determined by the last time step that the activation of a control signal is dependent on the condition value $\mathcal{V}_c$. It is obvious that a condition value must be "remembered" by a controller during the reserved period and the number of status registers required to implement the controller is the maximum number of condition values reserved for any cycle during the execution.

For an n-time-step non-pipelined design controller using status registers, a *ring state transition diagram* $\mathcal{R}_n$ is necessary and sufficient to specify the control behavior because the status registers are used to "remember" the propagated condition values. The control specification for a state in which no condition value is alive is the same as the control specification in a design without conditional branches; i.e. a set of activated control actions is followed by a next-state assignment. For the state with only one condition value alive, an *if-then-else* statement is used to specify the conditional branches. Otherwise, a *case* statement will be used to specify a state with more than one condition value alive.

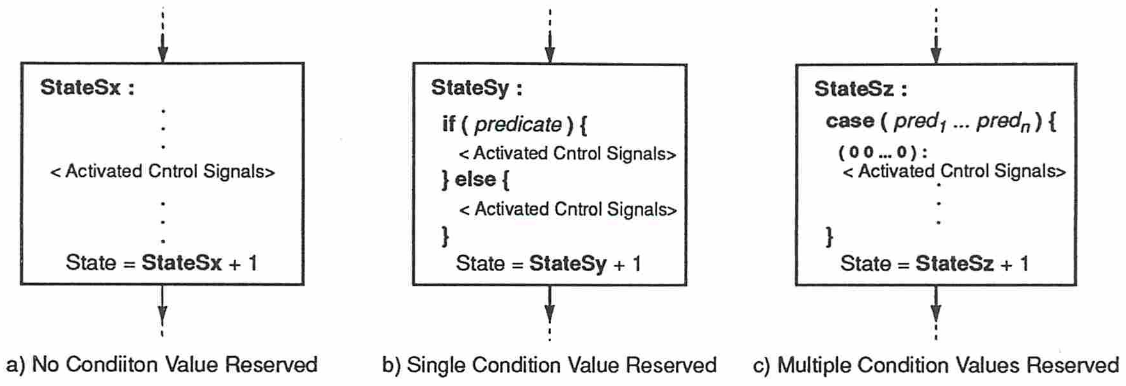
StateSx :
< Activated Cntrol Signals>
State = StateSx + 1
StateSy :
if (predicate) {
< Activated Cntrol Signals>
} else {
< Activated Cntrol Signals>
}
State = StateSy + 1
StateSz :
case (pred1 ... predn) {
(0 0 ... 0) :
< Activated Cntrol Signals>
}
State = StateSz + 1
a) No Condiiton Value Reserved
b) Single Condition Value Reserved
c) Multiple Condition Values Reserved

Figure 3: Control Specifications Using Status Registers

We now decide which condition value reserved is assigned to which status register. The Left Edge algorithm [7], which was proposed by Kurdahi to solve register allocation in data path synthesis, is used to assign each condition value reserved to a status register based on the results of condition-value lifetime analysis. A controller for a non-pipelined design with conditional branches can therefore be constructed similar to a controller for a design without conditional branches. The activated control actions for each state are specified by the templates shown in Figure 3. The reserved condition values are propagated according to the pre-determined status register assignments. A *ring state transition diagram* $\mathcal{R}_n$ is necessary and sufficient to specify the controller for an n-time-step non-pipelined design. The first state of the state transition diagram is the state of time step 1.

A controller for a pipelined design using status registers can resemble a non-pipelined design using status registers. The idea of a *folded* scheduled control/data flow graph is used to model the pipelined designs as above. We start by viewing the folded control/data flow graph as a set of *p-executions*. The purpose of an initiation-interval-I pipelined design controller is to issue the respective control signals, which are necessary to execute the operations in the *p-execution* $\mathcal{E}_{pi}$ for $i = 1, \ldots, I$, and for any cycle $t = mI + i$. Because status registers are used to save the condition value reserved, a controller approach similar to a pipelined design without conditional branches can be used here.

The Left Edge algorithm assigns the reserved condition values of a folded control/data flow graph to status registers in a similar way as above. However, a condition value may have several instances reserved during a single time step in a folded control/data flow graph due to the overlapped execution characteristic of pipelined designs. In that case, a condition value instance is moved between status registers according to the execution of the pipeline. A *ring state transition diagram* $\mathcal{R}_I$ is necessary and sufficient to specify a controller for a pipelined design with initiation interval I. The set of activated control signals in state $\mathcal{S}_i$ is the union of the control signals activated in time step $i, i + I, i + 2I, \ldots, i + mI$,which is similar to pipelined designs without conditional branches. The first state of the state transition diagram is determined by assigning the state of time step 1.

4.4 Controllers Without Status Registers

For designs with conditional branches, we now present a controller implementation using a state variable to store the reserved condition values. As compared to a controller using status registers, a controller without status registers usually has a more complicated state transition diagram. To construct a state transition diagram, we first determine the number of states which are required to represent the control behavior. From our experiences with controllers using status registers, a condition state should be saved by a controller during the *reserved period.* Obviously, there are *at most* 2^k states required to specify the controller for time step i if k condition values are reserved in time step i, where $1 \leq i \leq n$ for an n-time-step non-pipelined design. In practice, a design usually does not need so many states to completely specify the control behavior.

The determination of the next-state transitions of a current state is based on the results of condition-value lifetime analysis. In order to store reserved condition values in the state memory, we obtain the following. A state $\mathcal{S}_{i,p}$ in time step i has exactly one possible state transition, $\mathcal{S}_{i+1,q}$, in time step $i+1$ if there is no condition value born in time step i, or the condition values born in time step i are also dead in time step i. Also, a state $\mathcal{S}_{i,j}$ in time step i has 2 possible state transitions, $\mathcal{S}_{i+1,p}$ and $\mathcal{S}_{i+1,q}$, in time step $i+1$ if and only if there is one condition value $\mathcal{V}_c$ which is born in time step i and reserved in time step $i+1$, where $1 \leq i \leq n$ for an n-time-step non-pipelined design. The conditions of two states transiting to one state can be derived in a similar manner. Notice that one state is sufficient to specify the controller in time step 1 for an n-time-step non-pipelined design.

The procedure to construct the state transition diagram for a non-pipelined design starts by determining a set of states which is sufficient to represent control behavior for each time step. Each state is then assigned a unique condition value combination which indicates the executing conditions of that state, i.e. the state will be visited as long as all the condition values specified in the associated condition value combination are satisfied. The set of states is obtained either by color labels [5] or by assigning to 2^k states, whichever is smaller, where k is the number of condition values reserved in time step i.

Once the set of states is determined, the next state transitions for each generated state are decided by the results of condition-value lifetime analysis as mentioned above. The activated control signals are obtained based on the associated condition value combination in each state. Finally, we assign the first state to the state for time step 1 (note that only one state is generated for time step 1) and the construction of a state transition diagram for a non-pipeline design is completed.

The approach of controller design for a pipelined design is analogous to the controller design for a non-pipelined design. To clarify the resemblance, the *folded* scheduled control/data flow graph is used to show the overlapped executions of a pipelined design. Also because of the overlapped execution characteristics of a pipelined design, a condition value is treated as different condition value instances in each time step and should be stored separately by the controller, i.e. a condition value instance is actually a function of the condition value itself as well as its working time step.

To determine a set of states which is sufficient to represent the control behavior of a pipelined design, we start by generating a set of states which is sufficient to represent the control behavior for each time step without considering the overlapped execution characteristics of a pipelined design. Each state is associated with a condition value combination. This step is the same as we did for a non-pipelined design. Because a condition value is a function of the condition value itself and its working time step, each set of states before the control/data flow graph is folded, is mutually exclusive. The set of states, which is sufficient to represent a time step in the folded control/data flow graph, is obtained by the Cartesian Product of all the sets of states which are folded into this time step.

After the set of states for each folded time step is determined, the next state transitions of a state are decided similarly, except that a condition value is distinguished by the value itself as well as the working time step. The activated control signals are obtained by taking

1.2μm Technology	*Delay (ns)*	*Area (μm^2)*
16-bit Comparator (>)	36	142087
16-bit Adder (+)	34	68825
16-bit Subtractor (−)	36	93646
16-bit Multiplier (∗)	48	1838537

Table 1: Design Library Set Obtained from Cascade Design Automation ChipCrafter

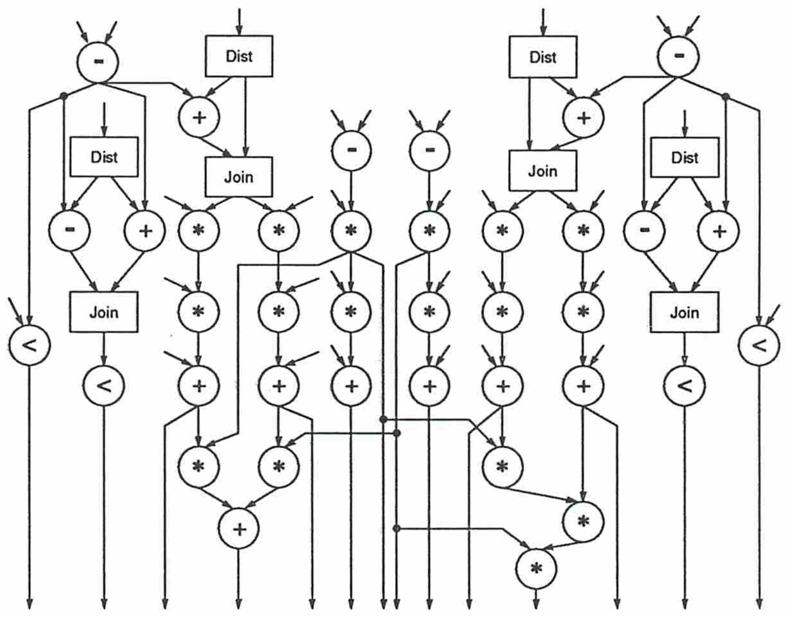

Figure 4: The Control/Data Flow Graph for a Robot Arm Controller

the union all the control signals activated in this folded time step. By assigning any one of the states in the folded time step 1 to be the first state, the state transition diagram of a pipelined design is completed.

5 Experiments and Results

The algorithms described in this paper have been implemented in a program called *CSG* using the C language. A portion of the robot arm controller example obtained from UC-Berkeley served as our example. The robot arm controller was originally written in the C language. The C code was translated into a VHDL description to obtain an internal control/data flow graph representation as shown in Figure 4. In this example, there are 46 operations/nodes including 4 distribute-join pairs. The module library used in our experiments is shown in Table 1.

Nine non-pipelined designs, as shown in Table 2, were created using the *MAHA* and *MA-*

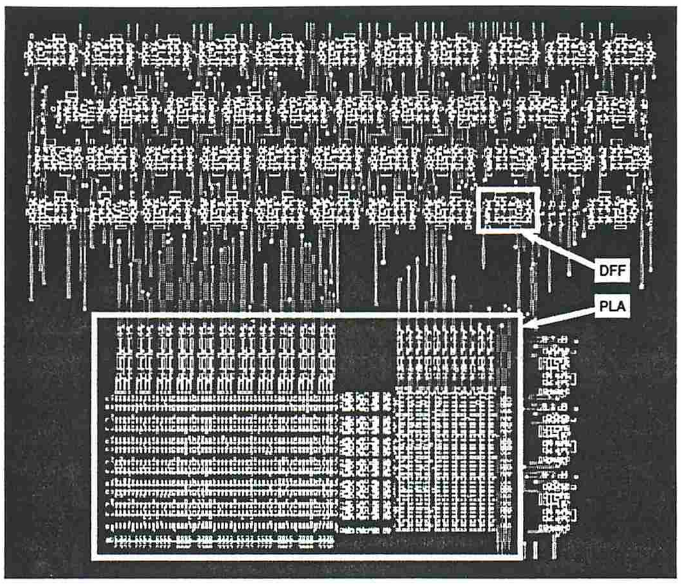

Figure 5: A PLA-Based Controller Synthesized by Finesse

Non-Pipelined	*Functional Units*				*Interconnection Units*	
Design Name[7]	*Cmp*	*Add*	*Sub*	*Mul*	*Register*	*Multiplexers*
MAHA.11	2	3	2	3	34	25
MAHA.12	2	2	4	2	34	21
MAHA.13	1	2	3	2	34	20
MAHA.14	1	1	3	2	34	15
MAHA.15	1	2	1	2	34	19
MAHA.19	2	1	3	1	34	14
MAHA.20	2	1	2	1	34	15
MAHA.21	1	1	2	1	34	12
MAHA.22	1	1	1	1	34	12

Table 2: Non-Pipelined Robot Arm Controller Examples by MAHA and MABAL

BAL programs to produce register-transfer level data paths. *CSG* then takes the schedule, bindings and register-transfer level data paths to generate the controller specifications. For each design, both types (with/without status registers) of controller specifications are produced and are shown in Tables 3 and 4, respectively. These designs are further synthesized by Cascade Design Automation ChipCrafter to obtain the layouts.

Figure 5 shows a typical controller layout produced by ChipCrafter which includes a PLA component and a set of D-type flip flops. It can be seen from this layout example that the area of the DFFs is almost the same as the PLA area. Wiring area also takes a substantial amount of area in a controller. Therefore, in order to estimate the area of a PLA-type controller accurately, the area of DFFs and wiring cannot be ignored.

Design Name	*States*	*S. Reg.s*	*Rows*	*Ipts*	*Opts*	*M. Opts*	*DFFs*
MAHA.11	12	2	17	10	39	45	42
MAHA.12	13	2	19	10	37	41	38
MAHA.13	14	1	18	9	44	36	52
MAHA.14	15	2	22	10	47	29	53
MAHA.15	16	2	24	10	48	32	50
MAHA.19	20	2	25	11	44	26	45
MAHA.20	21	2	26	11	49	24	50
MAHA.21	22	2	31	11	49	22	49
MAHA.22	23	2	30	11	47	23	48

Table 3: Controllers Using Status Registers for Non-Pipelined Examples

In those experiments, the size of PLAs in non-pipelined design controllers increased as the number of time steps increased (Figure 6). This behavior is consistent with our prediction. However, the area of the controllers increased in the beginning, but then fluctuated with less than 10% variation. This phenomenon may be explained as follows.

From Tables 3 and 4, the number of output signals decreased as the number of time steps increased in the controller specifications. Nevertheless, the number of merged outputs is reduced when the number of time steps is increased. This is probably because more output bits of a multiplexer are encoded into a select word for more serialized non-pipelined designs. For example, two 3-to-1 multiplexers (4 control bits) may be substituted for one 6-to-1 multiplexer (3 control bits) in a more serialized design. Even though the number of controller outputs (after removing the merged outputs), which is similar to the number of PLA outputs, is not increased in the designs with more than 12 time steps, the areas of the PLAs still increase. This is probably due to the fact that the number of PLA product terms(rows) and state encoding bits(columns) is increased as the number of time steps increases.

Figure 6 uses bar charts to show the area comparisons of PLAs and control paths for non-pipelined designs with/without status registers. It is interesting to note that a controller specified using status registers usually creates a smaller PLA area as well as smaller total controller area for non-pipelined designs, although it uses more state encoding bits[8] than one without status registers. In order to find out the reasons for this result, we examine the state transition diagrams for *MAHA.12* with/without status registers after Finesse[9]

[7]The naming convention for the non-pipelined designs is *MAHA*.[time steps].

[8]The state encoding bits here include the bit to represent states and the bits used by status registers.

[9]Finesse is a finite state machine synthesis program in Cascade Design Automation ChipCrafter.

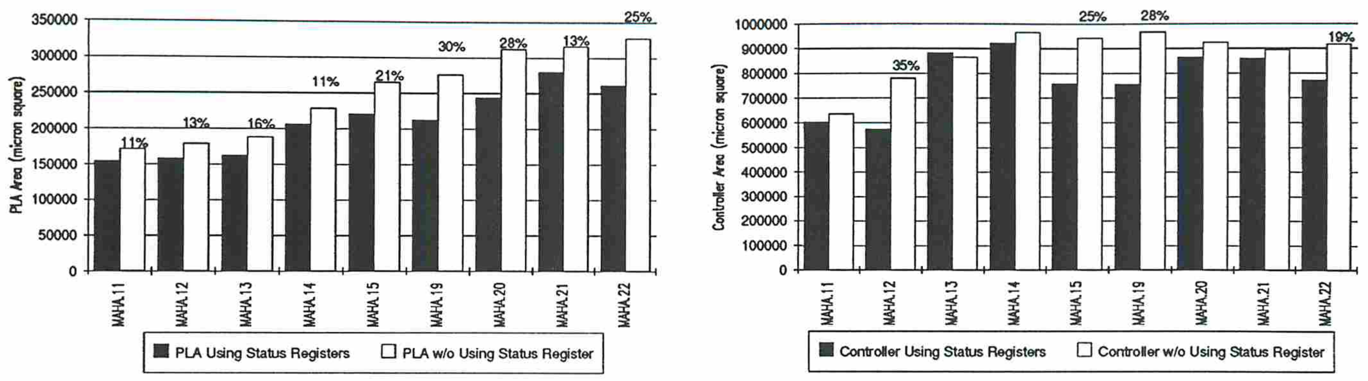

Figure 6: PLA and Controller Area Comparison of the Non-Pipelined Designs

Design Name	*States*	*Rows*	*Ipts*	*Opts*	*M. Opts*	*DFFs*
MAHA.11	21	21	9	40	44	40
MAHA.12	25	25	9	36	41	37
MAHA.13	19	19	9	47	34	50
MAHA.14	25	25	9	48	27	50
MAHA.15	30	29	9	50	29	51
MAHA.19	40	40	10	42	27	43
MAHA.20	41	39	10	46	26	49
MAHA.21	41	39	10	47	23	48
MAHA.22	42	41	10	46	23	47

Table 4: Controllers w/o Using Status Register for Non-Pipelined Examples

synthesis. Notice that a twelve-time-step design needs thirteen clock cycles to complete the task because one extra clock cycle is used to latch input values.

Figure 7 depicts the state transition diagrams for *MAHA.12* without/with status registers. The two state transition diagrams look different, however, they represent exactly the same controller behavior. By comparing these two state transition diagrams, we found that some of the states in the state transition diagram without status registers split into several states in the state transition diagram with status registers. All the split states are shown by bold face and cross hatch patterns in Figure 7. The splits of those states indeed indicate that a multiple-code state encoding is produced in the controller using status registers. Apparently, *Espresso* is able to exploit the don't care bits in state codes in order to minimize the size of the binary encoded cover. For example, no status register is used in *State0* for the controller using status registers, so, in Finesse, the two don't care bits and "0100" are assigned by the unicode state encoding algorithm. To reduce the product terms,

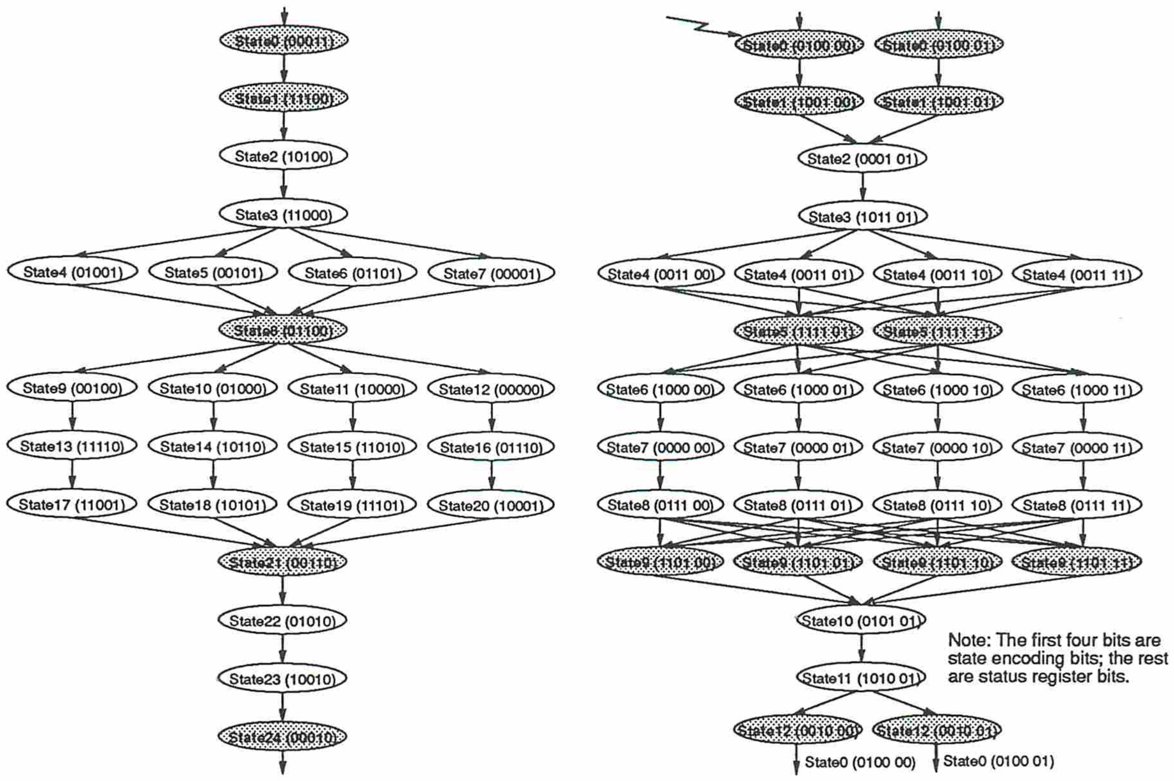

Figure 7: *MAHA.12* State Transition Diagram without/with Status Registers

Espresso assigned two don't care bits to "00" and "01" and the state encodings of *State0* became "0100 00" and "0100 01" (a multiple-code state encoding). Note that most current state encoding algorithms can produce only unicode encodings. Our heuristic functionally separates the encoding task between the states and the condition values, which make the unicode state encoding algorithm produce a multiple-code state encoding.

Fourteen pipelined designs with different initiation interval and/or pipeline length were created by *Sehwa* and *MABAL*, as shown in Table 5. Both the controller styles were generated for all the designs, as shown in Tables 6 and 7. To compare the two implementation styles, the bar charts for PLA area and controller area are depicted in Figure 8. The results are slightly different from the non-pipelined designs. In pipelined designs, some of the designs using status registers are larger than the ones without status registers. This may be due to the high utilization of status registers which leaves *Espresso* with no or little freedom to assign the don't care bits in status registers. However, the number of improved cases is still more than the number of worse cases in our experiments. Notice that the percentages

[10]The naming convention for pipelined designs is *SEHWA*.[pipeline length].[initiation interval].

Pipelined Design Name[10]	*Functional Units*				*Interconnection Units*	
	Cmp	*Add*	*Sub*	*Mul*	*Register*	*Multiplexers*
SEHWA.15.3	2	4	2	6	110	40
SEHWA.17.3	2	4	2	6	110	36
SEHWA.13.4	1	3	2	5	84	28
SEHWA.15.4	1	3	2	5	76	35
SEHWA.11.5	1	3	2	4	55	26
SEHWA.12.5	1	3	2	4	60	29
SEHWA.19.6	1	2	1	3	71	20
SEHWA.24.6	1	2	1	3	92	26
SEHWA.15.9	1	2	1	2	42	22
SEHWA.16.9	1	2	1	2	45	22
SEHWA.14.11	1	1	1	2	35	19
SEHWA.20.11	1	1	1	2	41	20
SEHWA.20.17	1	1	1	1	36	15
SEHWA.27.17	1	1	1	1	42	16

Table 5: Pipelined Robot Arm Controller Examples by SEHWA and MABAL

Design Name	*States*	*S. Reg.s*	*Rows*	*Ipts*	*Opts*	*M. Opts*	*DFFs*
SEHWA.15.3	3	4	11	10	13	163	21
SEHWA.17.3	3	2	6	6	11	159	14
SEHWA.13.4	4	3	10	8	14	120	24
SEHWA.15.4	4	1	8	7	14	125	26
SEHWA.11.5	5	1	9	8	26	79	33
SEHWA.12.5	5	2	11	9	22	92	29
SEHWA.19.6	6	1	11	7	31	87	39
SEHWA.24.6	6	4	22	10	40	111	42
SEHWA.15.9	9	1	13	9	45	48	48
SEHWA.16.9	9	2	17	10	50	46	57
SEHWA.14.11	11	1	18	10	51	30	53
SEHWA.20.11	11	2	19	9	48	44	54
SEHWA.20.17	17	2	23	11	47	29	50
SEHWA.27.17	17	2	23	11	49	35	51

Table 6: Controllers Using Status Registers for Pipelined Examples

Design Name	*States*	*Rows*	*Ipts*	*Opts*	*M. Opts*	*DFFs*
SEHWA.15.3	36	30	10	19	157	26
SEHWA.17.3	9	6	6	7	163	10
SEHWA.13.4	18	19	9	25	111	32
SEHWA.15.4	8	8	7	14	125	26
SEHWA.11.5	9	11	8	24	81	32
SEHWA.12.5	14	13	8	26	87	32
SEHWA.19.6	10	9	7	32	86	38
SEHWA.24.6	52	48	10	45	106	51
SEHWA.15.9	14	13	8	47	45	50
SEHWA.16.9	23	23	9	49	46	53
SEHWA.14.11	19	20	9	50	31	51
SEHWA.20.11	16	15	8	52	39	54
SEHWA.20.17	25	24	9	48	27	49
SEHWA.27.17	28	28	9	49	33	50

Table 7: Controllers w/o Using Status Register for Pipelined Examples

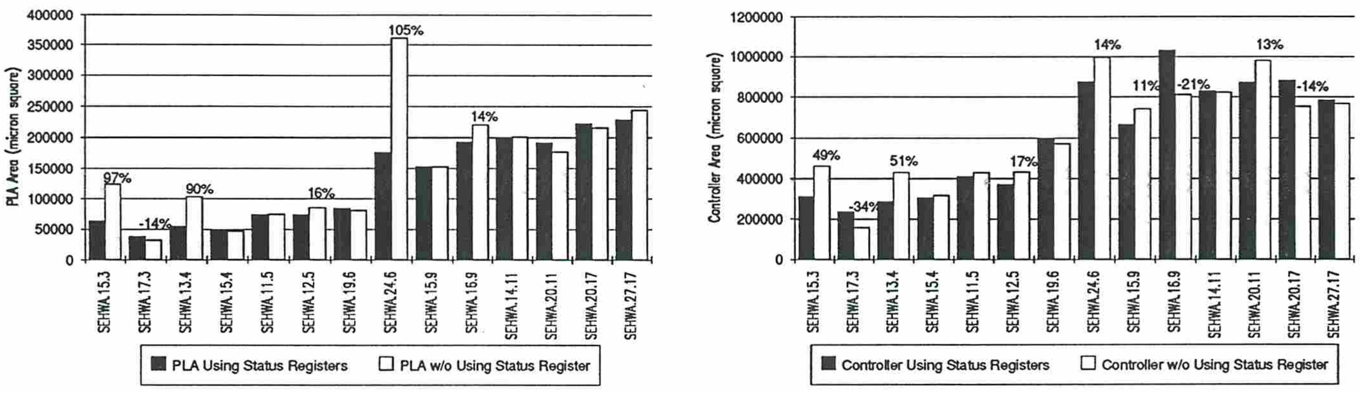

Figure 8: PLA and Controller Area Comparison of the Pipelined Designs

of area improvement of both PLA area and controller area are more dramatic than the ones for non-pipelined designs, and the percentage of area increase by the controllers using status registers is much less than the ones without status registers.

6 Conclusions and Future Work

In this paper, the control path synthesis problem is addressed. Two possible controller implementation strategies are proposed: controllers with status registers and without status registers. Intuitively, controllers without status registers would seem better, because less state encoding bits are needed. However, the experimental results contradict our initial guess (expectation): the controllers implemented using status registers are better results in all non-pipelined design cases and most pipelined design cases. After we traced one of non-pipelined design cases, we found out the improvement is caused by the result of a multiple-code state encoding instead of a conventional unicode state encoding. Therefore, the controllers implemented using status registers are able to produce better designs after logic simplification.

In general, a multiple-code state encoding, if done intelligently, is expected to produce better results then a unicode state encoding. However, to the best of our knowledge, there is no reported work on multiple-code state encoding algorithms among current state encoding research. One possible reason is the high complexity of computing optimal multiple-code state encodings. The controller implementation which uses status registers does provide an effective heuristic for producing multiple-code state encodings using a unicode state encoding algorithm.

There is still much work to be done here. More research on controller implementations with status registers versus without status registers may lead to more concrete results which may help us choose a better controller implementation style in the early stage of control path synthesis. This research will also help us develop more powerful multiple-code state encoding algorithms by giving hints to unicode state encoding algorithms. We believe that with further research on multiple-code state encodings we may be able to derive a more powerful and efficient heuristic for the state encoding problem.

References

[1] C. Chu et al. Hyper: An Interactive Synthesis Environment for High Performance Real Time Applications. In *Proceeding of 1989 IEEE Int. Conf. on Computer Design*, pages 432–435, October 1989.

[2] D. Gajski, editor. *Silicon Compilation*, chapter 8, pages 311–360. Addison-Wesley Publishing Company, Inc., 1988.

[3] D. Gajski, editor. *Silicon Compilation*, chapter 7, pages 204–310. Addison-Wesley Publishing Company, Inc., 1988.

[4] G. Goossens et al. An Efficient Microcode Compiler for Custom DSP Processors. In *Proceeding of 1987 IEEE Int. Conf. on Computer-Aided Design*, pages 24–27, November 1987.

[5] J. Kim and F. Kurdahi. Synthesis of Time-Stationary Controllers for Pipelined Data Paths. In *Proceeding of 1991 IEEE Int. Conf. on Computer-Aided Design*, pages 30–33, November 1991.

[6] D. Knapp. *A Planning Model of the Design Process.* PhD thesis, University of Southern California, December 1986.

[7] F. Kurdahi and A. C. Parker. REAL : A Program for REgister ALlocation. In *Proceeding of 24th ACM/IEEE Design Automation Conference*, pages 210–215, June 1987.

[8] A. Nagle, R. Cloutier, and A. Parker. Synthesis of Hardware for the Control of Digital Systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, CAD-1(4):201–212, October 1982.

[9] N. Park and A. Parker. Sehwa: A Software Package for Synthesis of Pipelines from Behavioral Specifications. *IEEE Transactions on Computer-Aided Design*, 7(3):356–370, March 1988.

[10] A. Parker et al. Experience with the ADAM Synthesis System. In *Proceeding of 26th ACM/IEEE Design Automation Conference*, pages 56–61, July 1989.

[11] C. Tseng et al. A Versatile Finite State Machine Synthesizer. In *Proceeding of 1986 IEEE Int. Conf. on Computer-Aided Design*, pages 206–209, November 1986.

[12] C. Tseng et al. Bridge: A Versatile Behavioral Synthesis System. In *Proceeding of 25th ACM/IEEE Design Automation Conference*, pages 415–420, June 1988.

[13] J. Weng and A. C. Parker. CSG : A Control Path Synthesis Program. Technical report, Department of Electrical Engineering, University of Southern California, 1992. In Progress.