Transformations Supporting Interactive Rescheduling for High-Level Synthesis

Roger Ang and Nikil Dutt

Technical Report 92-20
revised April 23, 1992

Department of Information and Computer Science
University of California, Irvine
Irvine, CA 92717
(714) 856-8059

rang@ics.uci.edu

Abstract

Traditionally, high-level synthesis (HLS) has been a fully automatic process over which the user has had little or no control. To make HLS an acceptable methodology for expert designers, we need to allow for more interactivity during synthesis. Since the scheduling step in HLS often determines the scope and quality of the ensuing synthesis tasks, we describe behavior-preserving transformations for manual rescheduling of behavior. We present the Structured Finite State Machine (SFSM) design model for scheduled behavior, show its equivalence to the behavioral Control-Data Flow Graph (CDFG), define primitive behavior-preserving transformations and indicate the utility of these transformations. The manual rescheduling capability we describe allows expert designers to alter an automatically generated schedule to overcome simplifications and assumptions made by automatic scheduling algorithms.

Contents

1 Introduction

Traditionally, high-level synthesis has been a fully automatic process over which the user has had little or no control, particularly in the selection of transformations and in the modification of synthesized designs. To make high-level synthesis an acceptable methodology for expert designers, we need to allow for more interactivity during synthesis. This paper describes behavior-preserving transformations that permit manual rescheduling of behavior, given the original behavior and a complete schedule.

The original behavior, typically written in an imperative HDL, is represented as a Control/Data Flow Graph (CDFG). After the scheduling step of high-level synthesis, the scheduled behavior corresponds to an abstract finite state machine, in a model we call the Structured Finite State Machine (SFSM). In order to preserve behavior during manual rescheduling, we need to maintain consistency between the rescheduled design (i.e., the SFSM) and the original behavior (i.e., the CDFG). We show the equivalence of the SFSM and scheduled CDFG models by providing a construction mechanism to go between these two models.

We then describe manual rescheduling transformations on the SFSM model. Since the initial schedule often determines the scope and quality of the ensuing high-level synthesis tasks, it is imperative to allow for user input after scheduling. Manual rescheduling may be useful for the following reasons:

1. HLS tools (including schedulers) often make simplified assumptions about functional units (e.g., delays, types, features such as pipelining).

2. HLS algorithms may use very abstract or unrealistic cost functions for scheduling. For instance, delays of operations, effects of physical design (placement, routing) on costs of scheduling, etc.

3. We may not know a priori the input data distribution which affects branching probabilities for each design, hence assumptions made by schedulers on input data may not model actual use.

4. Expert designers may have useful knowledge and "design tricks" that are specific to certain types of designs, and are thus often not encoded into a scheduler.

The manual rescheduling capability we describe in this report allows expert designers to alter an automatically generated schedule to overcome the simplifications and assumptions made by the scheduling algorithms, and also gives the designer freedom to alter automatically generated schedules to his/her liking.

2 Design Models and Notation

In order to lay the foundation for the behavior preserving transformations, we need to define the basic design models for the original (unscheduled) behavior and the behavior after scheduling.

2.1 Control/Data Flow Graph (CDFG)

A behavioral description in a hardware description language (HDL) is typically compiled into a variant of a Control-Data Flow Graph (CDFG) [OrGa86] [CaRo89] [CaTa89] [RuGa90] [DeJo91] . The CDFG is a representation that encapsulates information about the flow of control through a design description as well the flow of data for individual assignments in the description. It is derived from the flow graph concept popular in software compilers [AhSU86]. We define a CDFG as a directed graph, where:

$$\mathbf{CDFG} = (\mathbf{C}, \mathbf{E}, t, \mathbf{D}, \mathbf{I}, \mathbf{O}, o)$$

C: a set of control flow nodes.
E: a set of directed edges that connect the control flow nodes.
t: a transition function which maintains sequencing of the control flow nodes.
D: sets of data flow graphs.
I: a set of design inputs.
O: a set of design outputs.
o: an output function for the design.

These sets and functions are further defined below.

2.1.1 Control Flow Graph

We begin by defining the Control Flow Graph.
For the following let:

$\mathbf{C}_i$ be a node in the control flow graph.

$\mathbf{D}(\mathbf{C}_i)$ be the set of data flow graphs associated with node $\mathbf{C}_i$.

$indegree(\mathbf{C}_i)$ be the number of edges connected to $\mathbf{C}_i$ that are directed toward $\mathbf{C}_i$.

$outdegree(\mathbf{C}_i)$ be the number of edges connected to $\mathbf{C}_i$ that are directed away from $\mathbf{C}_i$.

$successor(\mathbf{C}_1, \mathbf{C}_2)$ test that $\mathbf{C}_2$ is a successor of $\mathbf{C}_1$, i.e., whether there exists a set of edges which form a path from $\mathbf{C}_1$ to $\mathbf{C}_2$.

$\mathbf{succ}(\mathbf{C}_i)$ be the set of control flow nodes that are immediate successors of $\mathbf{C}_i$.

A *path* from a node, $\mathbf{C}_1$, to a successor node, $\mathbf{C}_2$, is a set of edges and nodes that leads to $\mathbf{C}_2$ such that each edge is traversed once. Transitions and the output generated by a control flow node is determined by the current inputs and the data flow graphs associated with that CF node:

$$(t : \mathbf{C} \times \mathbf{D} \times \mathbf{I} \mapsto \mathbf{E})$$
$$(o : \mathbf{C} \times \mathbf{D} \times \mathbf{I} \mapsto \mathbf{O})$$

A control flow node, $\mathbf{C}_i$, can be one of four types:

$$\forall \mathbf{C}_i \mid \mathbf{C}_i \in \mathbf{C}, \mathbf{C}_i \in \left\{ \begin{array}{l} \mathbf{C}_{block} \\ \mathbf{C}_{test} \\ \mathbf{C}_{join} \\ \mathbf{C}_{loop} \end{array} \right\}$$

Block : corresponds to a "basic block" of code. The set of actions in the block are performed sequentially without interruption of the flow of control, i.e., $\mathbf{D}(\mathbf{C}_i)$ is an ordered list. For this type of node, $indegree(\mathbf{C}_i) = 1$ and $outdegree(\mathbf{C}_i) = 1$. Transitioning to the next node occurs after execution of all actions for the node.

$\forall \mathbf{C}_i \mid \mathbf{C}_i \in \mathbf{C}_{block}, \mathbf{D}(\mathbf{C}_i) \neq \emptyset,$
$t(\mathbf{C}_i) \mapsto \mathbf{E}_i$

Test : these perform conditional branching. The transition function for these nodes depends upon the conditional expression associated with each node. Each edge from a **Test** node will have a value associated with it. The edge with the value equal to the result of the condition expression for the **Test** node indicates the next transition. For this type of node, $indegree(\mathbf{C}_i) = 1$ and $outdegree(\mathbf{C}_i) >= 2$.

$\forall \mathbf{C}_i \mid \mathbf{C}_i \in \mathbf{C}_{test}, \mathbf{D}(\mathbf{C}_i)$ is the condition expression,
$t(\mathbf{C}_i, \mathbf{D}(\mathbf{C}_i)) \mapsto \mathbf{E}_i$

Join : these complement the Test nodes. No actions or conditional expressions are associated with these nodes. For this type of node, $indegree(\mathbf{C}_i) >= 2$ and $outdegree(\mathbf{C}_i) = 1$.

$\forall \mathbf{C}_i \mid \mathbf{C}_i \in \mathbf{C}_{join}, t(\mathbf{C}_i) \mapsto \mathbf{E}_i$

Loop : this is a special case of a conditional branching node to indicate the entry/exit point of a cycle in the control flow graph. It is a combination Test/Join node. The transition function for these nodes depends upon the conditional expression associated with each node. For this type of node, $indegree(\mathbf{C}_i) = 2$ and $outdegree(\mathbf{C}_i) = 2$. One of the edges to the node is for initial entry into the loop, while the other edge to the node is for reiteration of the loop. One of the edges from the node is for exiting the loop and the other edge from the node transitions to the nodes within the loop.

$\forall \mathbf{C}_i \mid \mathbf{C}_i \in \mathbf{C}_{loop}, \mathbf{D}(\mathbf{C}_i)$ is the condition for the loop.
$t(\mathbf{C}_i, \mathbf{D}(\mathbf{C}_i)) \mapsto \mathbf{E}_i$

Exceptions to the above are the designated *start* node and an optional *stop* node. Both of these nodes may be of any of the above types. The start node is the first node in the control flow graph according to the order of evaluation. On beginning the evaluation of the control flow graph, the starting transition goes to this node. For a start node, $\mathbf{C}_{start}$, $indegree(\mathbf{C}_{start})$ may be zero. A stop node is a node with no outgoing edges, i.e., $outdegree(\mathbf{C}_{stop}) = 0$.

In addition, CDFGs possess an important property of *well-structuredness*, that make various kinds of global data analyses possible. For every Test node there is a corresponding successor Join node.

$$\forall \mathbf{C}_i \mid \mathbf{C}_i \in \mathbf{C}_{test},\ \exists \mathbf{C}_j \mid \mathbf{C}_j \in \mathbf{C}_{join},\ successor(\mathbf{C}_i, \mathbf{C}_j)$$

Test-Join pairs must be properly nested. That is, let $\mathbf{C}_1$ and $\mathbf{C}_2$ be test nodes with corresponding join nodes $\mathbf{J}_1$ and $\mathbf{J}_2$, respectively. If $\mathbf{C}_2$ is on a path from $\mathbf{C}_1$ to $\mathbf{J}_1$, $\mathbf{J}_2$ must also be on the same path.

Let $\mathbf{C}_i$, $\mathbf{C}_j \in \mathbf{C}_{test}$, $\mathbf{C}_x$, $\mathbf{C}_y \in \mathbf{C}_{join}$, where $(\mathbf{C}_i, \mathbf{C}_x)$ and $(\mathbf{C}_j, \mathbf{C}_y)$ are test/join pairs.
If $successor(\mathbf{C}_i, \mathbf{C}_j)$, then $successor(\mathbf{C}_y, \mathbf{C}_x)$

For loops, any node within a cycle, except the Loop node, will have no edges that transition to a node outside of the cycle.

Let $\mathbf{C}_{cycle}$ be the set of CF nodes within a cycle in a CDFG.
$\forall \mathbf{C}_i \mid \mathbf{C}_i \in \mathbf{C}_{cycle}, \mathbf{C}_i \notin \mathbf{C}_{loop}$
$\forall \mathbf{C}_j \mid \mathbf{C}_j \in \mathbf{succ}(\mathbf{C}_i), \mathbf{C}_j \in \mathbf{C}_{cycle}$.

A loop node is the only entry and exit point for a cycle. If a Test node, $\mathbf{C}_i$, is in a cycle, its corresponding Join node, $\mathbf{C}_j$, must be also be in that cycle. Conversely, if $\mathbf{C}_i$ is a successor to a node $\mathbf{C}_l$ and $\mathbf{C}_i$ is not in a cycle, then $\mathbf{C}_j$, the Join node for $\mathbf{C}_i$, must be a successor of $\mathbf{C}_i$ and $\mathbf{C}_l$ and not be in a cycle.

To simulate the behavior described by a CDFG, control flow through the graph proceeds sequentially, i.e., only one node is under evaluation at any time. Upon encountering a node during evaluation, if there are any actions or conditions associated with the node, they are evaluated. After evaluation of actions or conditions, the transition function is called to indicate which edge from the current node should be followed to arrive at the next node for evaluation. The structure of the control flow graph determines the evaluation order of the nodes. The control flow graph structure along with the actions and conditions of nodes are the representation of a desired sequential behavior.

2.1.2 Data Flow Graph

Each control flow node $\mathbf{C}_i$ may have an associated data flow graph. A single data flow graph is a directed acyclic graph defined as:

$$\mathbf{DFG} = (D,\ E)$$

D: a set of data flow nodes.
E: a set of directed edges that connect the nodes.

For the following let:
D_i be a node in the data flow graph.
$indegree(D_i)$ be the number of edges connected to D_i that are directed toward D_i.
$outdegree(D_i)$ be the number of edges connected to D_i that are directed away from D_i.

There are two basic types of data flow nodes:

$$\forall D_i \mid D_i \in D, D_i \in \left\{ \begin{array}{l} D_{data\_access} \\ D_{operation} \end{array} \right\}$$

Data Access : we associate the name of a variable or data carrier with this type of node. The value of this variable or data carrier is accessed by this node. If an edge is directed towards this type of node, the variable is being assigned a value, i.e., "write" to the node. If an edge is directed from this type of node, the value of the variable is made available to the node directed towards, i.e., "read" from the node.

$$\forall D_i \mid D_i \in D_{data\_access}, D_i \in \left\{ \begin{array}{l} D_{read} \mid indegree(D_i) = 0, outdegree(D_i) > 0 \\ D_{write} \mid indegree(D_i) = 1, outdegree(D_i) = 0 \end{array} \right\}$$

Operation : we associate an operation with this type of node, e.g., arithmetic, logical and relational. The operation node receives data values from edges directed towards it, and passes on resulting data values via edges directed away from it. This type of node takes the values given to it, performs its associated operation on the values, and outputs the resulting value. The number of input values required, i.e., edges directed towards the node, depends upon the operation associated with the node.

A data flow graph can be used to represent an assignment statement found in a typical programming languages, e.g., a $\leftarrow$ b + 1. During the evaluation of a data flow graph, data can be considered to "flow" from the read nodes of the graph, through the operation nodes, and into the write nodes.

2.2 Structured Finite State Machine (SFSM)

After scheduling, we can represent a design by a finite state machine (FSM) with associated datapath operations: in each state of the FSM, certain data path operations are performed. FSMs are practical for logic and register-transfer synthesis because methods to implement them are well established. However, the original behavior comes from an imperative, well-structured HDL, which is compiled into a well-structured CDFG. We use a form of finite state machine, which we call a Structured Finite State Machine (SFSM), to bridge differences between the previously described CDFG model and traditional finite state machine models. The SFSM model is used to maintain the control flow structure of a scheduled CDFG within a finite state machine model. We show rules for constructing an SFSM from a scheduled CDFG and vice-versa, thereby establishing their equivalence. This gives us the advantage of performing data analyses on well-structured CDFGs, while using the SFSM for subsequent high-level synthesis tasks such as component allocation and bindings.

SFSMs are a subset of FSMs with datapath (FSMD) [GDWL92]. Briefly, FSMDs are finite state machines defined on abstract variables and their semantics to describe conditions for transitions, and assignments for transformation of inputs to outputs. FSMDs use data abstraction to avoid explicitly describing the binary/boolean encoding of data. An FSMD is thus equivalent to a scheduled behavior derived from an imperative HDL, where operations on abstract variables are performed in each state. An FSMD consists of:

$$\mathbf{FSMD} = (\mathbf{S}, \mathbf{I} \cup \mathbf{STAT}, \mathbf{O} \cup \mathbf{A}, f, h), \text{ where}$$

S: a set of states.
I: a set of inputs.
STAT: a set of status signals each consisting of a logical relational expression.
O: a set of outputs.
A: a set of assignments.
f: transition function. ($f: \mathbf{S} \times \mathbf{I} \cup \mathbf{STAT} \mapsto \mathbf{S}$)
h: output function. ($h: \mathbf{S} \times \mathbf{I} \cup \mathbf{STAT} \mapsto \mathbf{O} \cup \mathbf{A}$)

An SFSM is a well-structured form of FSMD. In an SFSM, there can be four type of states: Action, Condition, Join, and Loop Test.

$$\forall \mathbf{S}_i \mid \mathbf{S}_i \in \mathbf{S}, \mathbf{S}_i \in \left\{ \begin{array}{l} \mathbf{S}_{action} \\ \mathbf{S}_{condition} \\ \mathbf{S}_{join} \\ \mathbf{S}_{looptest} \end{array} \right\}$$

The semantics and usage of SFSM constructs can be best described in terms of the CDFG. An SFSM can be constructed from a scheduled CDFG in the following manner:

For each Test, Join or Loop node in the control flow graph of the CDFG, construct a corresponding state in the SFSM. If there is a condition expression associated with a Test or Loop node, construct an identical condition expression for the corresponding state.

For each Block node in the control flow graph of the CDFG, a corresponding series of states will be constructed in the SFSM. A Block node in a CDFG has of a sequence of actions associated with it. Data dependencies between the actions may require that some actions be evaluated before others. This means that the actions may have to be performed in a series of steps. Actions with no data dependencies between them may be performed in the same step.

Assigning actions to control steps so as to best meet the clock cycle time is called "scheduling". The number of states needed in the SFSM to model a Block node from the CDFG is equal to the number steps required for the actions in the Block node. That is, for each state scheduled to a CDFG Block node, there is a corresponding state in the SFSM. For each state in a scheduled Block node, the actions for that state are copied to the corresponding SFSM state. Since only actions with no data dependencies between them may be scheduled to the same state, the actions in a state of the SFSM can be considered as occurring concurrently.

For each edge in the CDFG, a corresponding transition is constructed in the SFSM. For CDFG nodes $\mathbf{C}_i$ and $\mathbf{C}_j$ with corresponding SFSM states $\mathbf{S}_i$ and $\mathbf{S}_j$, respectively, if there is an edge from $\mathbf{C}_i$ to $\mathbf{C}_j$, a transition from $\mathbf{S}_i$ to $\mathbf{S}_j$ is constructed. If a condition value is associated with the CDFG edge, an identical value is associated with the SFSM transition. Transitions to and from Block nodes are special cases since they map to a series of Action states in the SFSM. An edge to a Block node in a CDFG is modeled as a transition to the first state of a series of Action states. Likewise, an edge from a Block node is modeled as a transition from the last state of a series of Action states.

Let

$STAT(\mathbf{S}_i)$ be the set of condition expressions associated with state $\mathbf{S}_i$.
$A(\mathbf{S}_i)$ be the set of assignments associated with state $\mathbf{S}_i$.

Given a CDFG, $\mathbf{X}_{cdfg} = (\mathbf{C}, \mathbf{E}, t, \mathbf{D}, \mathbf{I}, \mathbf{O}, o)$,
construct an SFSM, $\mathbf{X}_{sfsm} = (\mathbf{S}, \mathbf{I} \cup \mathbf{STAT}, \mathbf{O} \cup \mathbf{A}, f, h)$,
such that $\mathbf{X}_{cdfg} \approx \mathbf{X}_{sfsm}$, i.e.,

$\forall \mathbf{C}_i \mid \mathbf{C}_i \in \mathbf{C}, \mathbf{C}_i \in \mathbf{C}_{test}, \exists \mathbf{S}_i \mid \mathbf{S}_i \in \mathbf{S}, \mathbf{S}_i \in \mathbf{S}_{condition}, STAT(\mathbf{S}_i) \approx \mathbf{D}(\mathbf{C}_i)$

$\forall \mathbf{C}_i \mid \mathbf{C}_i \in \mathbf{C}, \mathbf{C}_i \in \mathbf{C}_{join}, \exists \mathbf{S}_i \mid \mathbf{S}_i \in \mathbf{S}, \mathbf{S}_i \in \mathbf{S}_{join}$

$\forall \mathbf{C}_i \mid \mathbf{C}_i \in \mathbf{C}, \mathbf{C}_i \in \mathbf{C}_{loop}, \exists \mathbf{S}_i \mid \mathbf{S}_i \in \mathbf{S}, \mathbf{S}_i \in \mathbf{S}_{looptest}, STAT(\mathbf{S}_i) \approx \mathbf{D}(\mathbf{C}_i)$

$\forall \mathbf{C}_i \mid \mathbf{C}_i \in \mathbf{C}, \mathbf{C}_i \in \mathbf{C}_{block}, \exists \{\mathbf{S}_i, \cdots, \mathbf{S}_j\} \subseteq \mathbf{S} \mid \{\mathbf{S}_i, \cdots, \mathbf{S}_j\} \subseteq \mathbf{S}_{action}, \{A(\mathbf{S}_i) \cup \cdots \cup A(\mathbf{S}_j)\} \approx \mathbf{D}(\mathbf{C}_i)$
where the number of states $\mathbf{S}_i, \cdots, \mathbf{S}_j$ is equal to the number of steps (control) needed to evaluate $\mathbf{D}(\mathbf{C}_i)$, and $\mathbf{S}_i, \cdots, \mathbf{S}_j$ is an ordered sequence of states.

$\forall \mathbf{E}_i \mid \mathbf{E}_i \in \mathbf{E}, \mathbf{E}_i$ is directed from $\mathbf{C}_i$ to $\mathbf{C}_j$, $f(\mathbf{S}_i, \mathbf{I} \cup STAT(\mathbf{S}_i)) \mapsto \mathbf{S}_j$

Exceptions:

- transitions to and from block nodes
  $\forall \mathbf{C}_x \mid \mathbf{C}_x \in \mathbf{C}_{block} \mid \mathbf{C}_x \approx \{\mathbf{S}_x, \cdots, \mathbf{S}_y\}, \mathbf{succ}(\mathbf{S}_x) = \mathbf{S}_{x+1}, \cdots, \mathbf{succ}(\mathbf{S}_{y-1}) = \mathbf{S}_y$
  if $\exists \mathbf{E}_i \mid \mathbf{E}_i \in \mathbf{E}$ that is directed from node $\mathbf{C}_i$ to node $\mathbf{C}_x$, then $f(\mathbf{S}_i, \mathbf{I} \cup STAT(\mathbf{S}_i)) \mapsto \mathbf{S}_x \mid \mathbf{C}_i \approx \mathbf{S}_i$
  if $\exists \mathbf{E}_i \mid \mathbf{E}_i \in \mathbf{E}$ that is directed from node $\mathbf{C}_x$ to node $\mathbf{C}_j$, then $f(\mathbf{S}_y, \mathbf{I} \cup STAT(\mathbf{S}_y)) \mapsto \mathbf{S}_j \mid \mathbf{C}_j \approx \mathbf{S}_j$
- stop state
  if $\mathbf{C}_i$ is a stop node with a corresponding state in an SFSM, $\mathbf{S}_i \mid \mathbf{C}_i \approx \mathbf{S}_i$,
  then $f(\mathbf{S}_i, \mathbf{I} \cup STAT(\mathbf{S}_i)) \mapsto \mathbf{S}_i$

From this construction, we see that the *well-structuredness* property for control flow nodes of a CDFG also apply to the states of an SFSM: Condition-Join pairs must be properly nested, cycles (loops) in an SFSM must be closed, and there is a "start" state and optional "stop" states. In a similar fashion, a reverse construction can be used to derive a CDFG design from an SFSM description. This proves the equivalence of the CDFG and SFSM models.

Note that, while a scheduled CDFG is modeled by an SFSM, states in the SFSM do not correspond directly to the scheduled states of the CDFG. That is, a state in the SFSM is not semantically the same as a control step in the CDFG since each control flow node in the CDFG is modeled by one or more states in the SFSM. While scheduling a CDFG, several control flow nodes can be assigned to the same control step (state).

We now show that the CDFG model is contained within the FSMD model. Since we proved the equivalence of the CDFG and SFSM models, we only need to describe the construction rules for converting an arbitrary SFSM to an FSMD, such that the states of the FSMD correspond directly to the states of the scheduled CDFG. The steps used in the construction of an FSMD from an SFSM are, in principle, similar to rules used in scheduling a CDFG. The construction of an FSMD from an SFSM consists of two basic steps:

1. *Merge Condition/Loop Test and Action States*: if a Condition (or Loop Test) state has immediate successors that are Action states, merge the action states with the Condition/Loop Test state. This is done by moving the actions to the Condition/Loop Test state and associating the condition for the transition to the Action state with the moved actions.

   Let `Ti` be a Condition or Loop Test state.

   ```
   for each Si in succ(Ti)
     if Si is an Action state then
       {
         move actions in Si to Ti
         associate stat(Ti) with actions where
         f(Ti, I U stat(Ti)) -> Si
       }
   ```

2. *Eliminate Join States*: Join states are artifacts of the original CDFG description which serve no purpose as far as behavior is concerned. They can be eliminated by "skipping" them in the FSMD model:

   Let `Ji` be a Join state.

   ```
   for each Si in pred(Ji)
      change edge to Ji to edge to succ(Ji)
   remove Ji and edges connected to it.
   ```

Consequently, given a SFSM model, a user can derive both a CDFG and an FSMD model that are closely related and behaviorally identical. Furthermore, we have proved the equivalence of scheduled CDFG and SFSM models, and showed that the scheduled CDFG (and hence the SFSM) model is contained within the FSMD design model.

Several advantages arise from using the CDFG and SFSM models. First, we formally defined the concepts of CDFG and SFSM and showed their equivalence. Hence, any changes made to the SFSM can be correlated to the behavioral CDFG to test for violations. Second, since a scheduled CDFG and SFSM are behaviorally equivalent, we can translate between the two representations for tasks appropriate to each design model. Third, by using a hybrid of flow graphs and FSMs, we are able to use both flow graph and FSM analysis.

2.3 Data Dependency Graph

Knowledge of the data-dependencies between actions in a description is crucial to preserving the functionality of the design. For an individual data carrier, such as a variable, the functionality of the data carrier is preserved as long as the order in which data is assigned and used is not altered. We use the concept of definition-use chaining (du-chaining) [AhSU86] to maintain this ordering. In the example, $x \leftarrow a + b$, the variable x is "defined" by the action and variables a and b are "used." In du-chaining, links are maintained between each use of a variable and the definitions that may apply to it. To represent du-chaining we use the notion of a data dependency graph:

$$\mathbf{DD} = (\mathbf{V},\ d,\ \mathbf{O},\ \mathbf{s})$$

V: a set of variable instances.
d: a dependency relation function.
O: a directed graph.
s: an ordering function, $\mathbf{V} \mapsto \mathbf{O}$.

A variable instance is a specific usage of a variable. For example, in the action $a \leftarrow a + b$ there are two instances of the variable a and one instance of variable b. The instance of a on the left of the arrow will be called a *definition* of a. While the instance of a on the right of the arrow will be called a *use* of a.

$$\forall v_i \mid v_i \in \mathbf{V}, v_i \in \left\{ \begin{array}{l} \mathbf{V}_{definition} \\ \mathbf{V}_{use} \end{array} \right\}$$

Each variable instance is associated with a node in the ordering graph, **O**. $s(v_i)$ returns the node in **O** that variable instance v_i is associated with. This mapping is used to specify a partial order on the variable instances **V**. An ordering graph **O** can be derived from a scheduled CDFG or an SFSM. The structure of the ordering graph is identical to that of the state transition graph of a given description, where the nodes correspond to states and the edges correspond to transitions. However, there must be no state in which a variable is defined twice, i.e., two definition instances in the same state. The variable instances of a given action are associated with the node in **O** that corresponds to the state the action was scheduled to.

for $\mathbf{S}_i \in \mathbf{S}$ of an SFSM, $\exists$ a node $s_i \in \mathbf{O}$ of the data-dependency graph.

for transition $f(\mathbf{S}_i, \mathbf{I} \cup STAT(\mathbf{S}_i)) \mapsto \mathbf{S}_j$ of an SFSM, $\exists$ an edge from s_i to s_j in **O** of the data-dependency graph, where $\mathbf{S}_i \mapsto s_i$ and $\mathbf{S}_j \mapsto s_j$

$\nexists v_i, v_j \mid v_i, v_j$ are definitions and $s(v_i) = s(v_j)$

The heart of the data-dependency graph is the dependency relation function, d. $d(v_i, v_j)$ returns true if there a data dependency between the variable instance v_i and v_j.

For the following let:

A *path* from a node s_1 to a node s_2 be a set of edges and nodes in **O** from s_1 to s_2 such that each edge is traversed once.

$successor(s_1, s_2)$ test that s_2 is a successor of s_1, i.e., whether there exists a path from s_1 to s_2.

$d(v_i,\ v_j)$ is true given:

$v_i, v_j \in \mathbf{V}$ are instances of the same name variable where v_i is a *definition* and v_j is a *use*.

(D1) $\exists$ a path from $s(v_i)$ to $s(v_j)$ (i.e., $successor(s(v_i), s(v_j))$).

(D2) $\nexists v_k \mid v_k$ is a *definition* of the same name variable and v_k is in a node on the path (i.e., $successor(s(v_i), s(v_k))$ and $successor(s(v_k), s(v_j))$)

3 Related Work

Similar application of compiler optimizations to tasks in high-level synthesis has been done before [BhLe90] [Camp90] [Walk82], but differ from this work in the design models and approach. Interactive use of similar transformations for sequential circuit behavior, **retiming**, has been done by [PoRa91] and [MSBS91]. However, **retiming** differs fundamentally from **rescheduling** in the underlying data being transformed. **Retiming** tools manipulate register-transfer (RT) level descriptions of circuits. Typically for **retiming**, the connectivity of registers to operational units is manipulated so that the functionality of the circuit is not altered while other characteristics, such as resource utilization, minimum cycle time, etc., are explored. While these tools allow changing the usage and number of registers in a circuit, they will not change the usage or number of operational units. **Rescheduling** performs transformations on the design prior to allocation, binding, and connecting of functional units. It allows manipulation of abstract operations in a behavioral description of a circuit. Consequently, **rescheduling** influences the usage and number of RT units (registers and operational) for a circuit synthesized from a transformed behavioral description.

[LeWo91] and [WaTh91] have explored interactive transformation of behavioral descriptions similar to this work. However, the system [LeWo91] developed is based entirely on FSMs. A given behavioral description is a nondeterministic FSM(BFSM) which is translated to a deterministic FSM(RTFSM). This RTFSM could in turn be transformed into other RTFSMs representing alternate schedules for the original behavior. An admitted shortcoming of this approach is that describing behaviors as state transition graphs is inconvenient. Translation of behaviors to BFSM from other formats, such as HDLs, introduces the problem of verification of equivalence of between the BFSM and HDL descriptions, which at this time has not been solved. We avoid this problem by using established compiler techniques to accurately map a VHDL description to a CDFG, and formulating an SFSM that is structurally equivalent and can be easily translated to and from a scheduled CDFG.

[WaTh91] have developed a system to manipulate value trace (VT) [McFa78] representations derived from ISPS. Their system performs code motion and condition merging transformations similar to the transformations described in this paper. However, the VT representation model is a data flow graph model variant. Unfortunately, data flow graphs are not amenable to some tasks such as state minimization or control analysis for resource sharing. For our work, we use a CDFG and formulated the SFSM for the very reason that these models will facilitate a wide range of synthesis tasks.

4 Primitive Operations

A. The behavior-preserving transformations used for our rescheduling scheme can be accomplished using a small set of primitive operations. By using these primitive operations in proper sequence, the transformations can manipulate the behavioral description while ensuring that the sequential behavior of the design is preserved. An essential primitive operation that will need to be performed by most of the transformations is a check of data dependencies. Given an action and a specified state, this primitive operation will indicate if placing the action within that state would violate the constraints of the data-dependency graph. Every action in a state must be independent of the other actions, so no data-dependencies can exist between any actions in a state. Consequently, the actions in a state can be performed concurrently. In the example below, if the definition of x in action the action $x \leftarrow y+1$ reaches the first action in state $\mathbf{S}_1$ (b $\leftarrow$ x), the data-dependency check would return that a conflict exists. In a transformation, this would abort the attempted change and the user would be notified that this transformation is not allowed.

$$dcheck\_action(x \leftarrow y+1,\ \mathbf{S}_1)$$

$$x \leftarrow y+1 \qquad\qquad \mathbf{S}_1 = \boxed{\begin{array}{c} b \leftarrow x \\ a \leftarrow 3 \end{array}}$$

B. A primitive operation that changes the description is the copying of an action to a state. This primitive operation is given an action and a specified state to copy the action to. It returns the specified state with the action inserted into it. In the transformations, this copy operation is only attempted when the previously described dependency check indicates no conflicts exist between the action and the state. Thus, the data-dependency graph, defined by the sequential behavior of the design, would not be violated and the functionality of the description is not altered.

$$copy\_action(x \leftarrow y+1,\ \mathbf{S}_1)$$

$$x \leftarrow y+1 \ \longrightarrow \mathbf{S}_1 = \boxed{\begin{array}{c} b \leftarrow x \\ a \leftarrow 3 \end{array}} \Longrightarrow \mathbf{S}_1 = \boxed{\begin{array}{c} b \leftarrow x \\ a \leftarrow 3 \\ x \leftarrow y+1 \end{array}}$$

C. Another primitive operation that alters the behavioral description is the deletion of an action from a state. This operation is given an action and a specified state to remove the action from. It returns the specified state with the action removed from it (see figure below). This operation should only be performed in conjunction with the copy operation to effect the movement of an action from one state to another. When used in this manner, this operation only removes a redundant action and no change is made to the sequential behavior of the design.

$$delete\_action(x \leftarrow y+1,\, \mathbf{S}_0)$$

$$\mathbf{S}_0 = \boxed{\begin{array}{c} z \leftarrow y-2 \\ x \leftarrow y+1 \end{array}} \Longrightarrow \mathbf{S}_0 = \boxed{z \leftarrow y-2}$$

D. In order to effect the moving of an action or subexpression, the detection of an action or subexpression is necessary. Given a state and an action/subexpression, this operation determines if the action/subexpression exists within the state. If so, it returns "handle(s)" to the occurrences of the action/subexpression. Action/subexpression detection is a form of pattern matching on a tree which was done in [Keut87]. In the model we use, expressions are represented as binary trees where internal nodes are operators and leaves are operands. In the case when the detection operation is searching for an action, the search involves matching both the variable that will get the resulting value and the expression tree for the right hand side expression. Matching of expression trees can be greatly simplified by applying a few rules to the construction of the trees, such as all trees will be right-derivations of an expression and operands will be ordered lexically. In the case of detecting a subexpression, a possible subtree must be matched. A brute force approach would be to search all permutations of an expression tree. However, the number of possible expression trees for an expression with $\mathbf{N}$ operators is on the order of $O(\mathbf{N}!)$. An alternative approach is to limit the search to the two or three levels of the expression trees closest to the root, i.e. permutation of the tree is done only on the first few applicable operations in the order of evaluation. This would mean, in the worst case, the search would go through only several dozen trees. While this search will find a common subexpression in an expression with 3 operators or less, it may not find subexpressions for expressions with 4 or more operators. This is a reasonable and practical approach, since we are dealing with expressions within a scheduled state, and scheduled states in a behavioral description typically do not have more than 3 chained operators. In practice, chaining more than 3 operators within a state leads to excessively long clock cycles and possibly poor resource utilization, as observed by Marwedel [Marw86].

E. After the occurrences of a subexpression have been found, the subexpression must be "factored" out of the "parent" actions before movement is possible. This factoring operation is illustrated below, where $b+x$ is the subexpression being factored out. This primitive operation is given an expression, an action, and an expression tree for the action. It returns two actions, one for the factored subexpression and one for the optimized parent action. In the transformations, this operation is used in conjunction with the copy and delete operations to move the new action containing the specified subexpression to a previous state, thus avoiding data-dependency conflicts. This operation updates the data-dependency graph, but from an external I/O view of the design, the sequential behavior is not altered.

$$factor(b+x,\, a \leftarrow b+x-c)$$

$$a \leftarrow b+x-c \Longrightarrow \begin{array}{c} tmp \leftarrow b+x \\ a \leftarrow tmp - c \end{array}$$

F. The number of states in the design can be changed by the insertion or deletion of empty states. Only action states can be added or deleted by this primitive operation. For the insertion operation, the location of where to insert the new state must be specified, i.e. between what states should the new state be inserted. For deletion, the specified state must have no actions and no condition tests. These operations will produce a design with an empty state added or removed. Since only empty states are inserted or deleted, these operations add no actions to the design nor change the order of actions. Consequently, these operations do not alter the sequential behavior of the design.

G. A single primitive operation will perform the merging of conditions. A requirement for merging conditions is that the conditions must be nested with no intervening states. There must be no states between the branching states and the joining state. The operation is given two adjacent condition states. It merges the conditions into a single condition state and, likewise, merge the join states. Conditions are limited to testing the value or particular bits of a variable. Since the test variables are set prior to testing and the tests have no side effects on any variables, merging conditions amounts to scheduling the tests of both conditions to a single state instead of two adjacent states. The variable values under which the various condition branches are performed are not changed. Therefore, the sequential behavior of a description is not altered by this operation.

5 Behavior-Preserving Rescheduling Transformations

Using the primitive operations from the previous section, we now describe the behavior-preserving rescheduling transformations. We use the following notation and functions.

(for the following `i` is an optional integer)
`Ai`: an action, such as a $\leftarrow$ b + 1
`SEi`: a subexpression, such as b + x
`Si`: a state
`ASi`: an action state
`Ci`: a condition state
`Ji`: a join state

Primitive operations
`dcheck_action(A, S)`, `dcheck_subex(SE, S)`
`copy_action(A, AS)`
`remove_action(A, AS)`
`detect_action(A, AS)`, `detect_subex(SE, AS)`
`factor_subex(SE, A)`
`insert_state(S, S)`
`delete_state(AS)`
`merge_conds(C, C)`

Functions
`pred(S)`: the set of states that are immediate predecessors of state S
`succ(S)`: the set of states that are immediate successors of state S

Knowledge of the data-dependencies between actions in the description is crucial to preserving the functionality of the design. For an individual data carrier, such as a variable, the functionality of the data carrier is preserved as long as the order in which data is assigned and used is not altered. The concept of definition-use chaining (du-chaining) [AhSU86] is used to maintain this ordering. In the example, $x \leftarrow a + b$, the variable x is "defined" by the action and variables a and b are "used." In du-chaining, links are maintained between each use of a variable and the definitions that may apply to it. The test variables in condition states are considered as "uses" of those variables. A data-dependency conflict occurs in the following cases:

1. a "use" and a "definition" that applies to it are in the same state and one of the instances was moved. This is necessary for maintaining the relative order of uses and definitions within loops.

2. more than one definition of a variable in a state. This ensures that the order of the values assigned to a variable is preserved

3. a use of a variable is moved beyond the scope of an applicable definition, e.g. into or before the state of a definition that applies to it, or into a state after a definition that does not apply to it.

4. a definition of a variable is moved into the scope of another definition, e.g. into a state before a use that the definition does not apply to, or into or after a state of use that the definition applies to.

The simplest transformation to perform on the design is the addition or deletion of an empty state (Figure 1). These transformations are the same as the primitive operations described in the previous section. The user specifies where a new state should be inserted, or which empty state to remove. Addition of an empty state will allow greater flexibility in re-ordering the existing actions. Though there is an increase in the number of states in the design, the re-ordering may make it possible to have a shorter clock cycle or a fewer number of functional units. The deletion of an empty state simply reduces the number of states in the design in hopes of reducing the number of clock cycles for the design, and consequently, improve its performance.

```
insert_state(S1, S2)
{
  if S1 is in pred(S2) then {
    Prev_State = S1
    Next_State = S2
  }
  else
    if S2 is in pred(S1) then {
      Prev_State = S2
      Next_State = S1
    }
    else abort transformation, specified states are not adjacent

  create new state S'
  make S' a next state for Prev_State
  make Next_State the next state for S'
}

delete_state(S1)
{
  if S1 has any actions or a condition then
    abort transformation, state not empty

  /* S1 should be an empty action state */

  find pred(S1)
  find succ(S1)
  make succ(S1) the next state for pred(S1)
  remove S1
}
```

Most of the other rescheduling transformations deal with the various cases of action movement. These transformations allow the user to re-order actions to change the states they are performed in. In these transformations the user specifies an action (or subexpression) in a particular state, and the destination state for the action (or subexpression).

The simplest form of action movement is moving an action/subexpression across other action states, i.e. movement does not go through a condition or join state (Figure 2). This is analogous to rearranging "straight-line code" in the behavioral description. This transformation is permitted if, for any state moved through or into, there are no data-dependency conflict between the action/subexpression and a state. A subexpression may only be moved "up" to a previous state. The steps involved in this transformation are outlined below:

1. check for data-dependency conflicts in each state along the path of movement
2. if moving a subexpression "up", for each state along the path of movement, including the source state but excluding the destination state:

   a. detect occurrences of the subexpression in the state

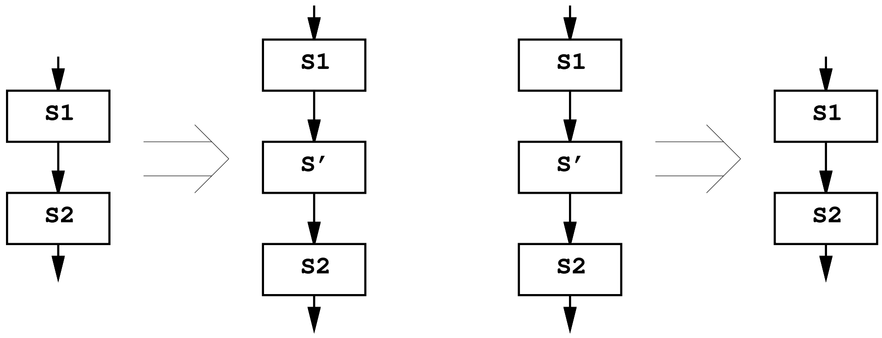

Figure 1: Insertion and deletion of an Action state.

b. if there is an occurrence, factor out the subexpression The new action containing the subexpression is the action being moved.

3. copy the action to the destination state.

4. if not moving a subexpression delete the action from the source state.

```
move_action_straight(AS1, AS2, A1)
{ /* AS1 is the source state, AS2 is the destination state */
  for each state Si from AS1 to AS2 {
     if Si is not an action state then
       abort transformation
     else
       if dcheck_action(A1, Si) returns a conflict then
         abort transformation
  }

  copy_action(A1, AS2)
  remove_action(A1, AS1)
}

move_subex_straight(AS1, AS2, SE1)
{ /* AS1 is the source state, AS2 is the destination state */

  if AS2 is not a predecessor of AS1 then
    abort transformation, subexpressions can only be moved up

  for each state Si from AS1 to AS2 {
     if Si is not an action state then
       abort transformation
     else
       if dcheck_subex(SE1, Si) returns a conflict then
         abort transformation
  }

  for each state ASi from AS1 to AS2
    for each Ai in detect_subex(SE1, ASi)  /* occurrences of SE1 */
```

```
    factor_subex(SE1, Ai)

  create a new action A2 for SE1
  copy_action(A2, AS2)
}
```

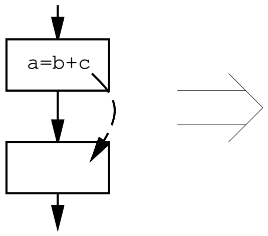

Figure 2: Action move to adjacent Action state.

When this transformation is combined with the addition and deletion of empty states, the combination performs the equivalent of state merging and splitting.

How an action is moved across a condition state depends on the direction of movement. For these transformations action states must immediately precede and succeed the condition state. When an action is moved "down" across a condition state, i.e. an action is moved across a successor state which is a condition state, the steps to accomplish this transformation are similar to the ones for simple straight-line movement. This transformation moves an action from the action state immediately preceding a condition state to the action states immediately succeeding that condition (Figure 3). Since it would not make sense, subexpressions cannot be moved in this way, i.e. subexpression cannot be propagated downwards. The algorithm is given below.

```
move_action_xcond_down(AS1, AS2, A1)
{ /* AS1 is the source state, AS2 is one of the destination states */
  if succ(S1) and pred(S2) are not the same condition state, C1, then
    abort transformation

  if dcheck_action(A1, C1) indicates a conflict then
    abort transformation

  for each state ASi in succ(C1) {
    if dcheck_action(A1, ASi) indicates a conflict then
      abort transformation
  }

  for each state ASi in succ(C1)
    copy_action(A1, ASi)
  remove_action(A1, AS1)
}
```

Movement of an action or subexpression "up" across a condition state has different steps than downward movement. This transformation moves an action from the action states immediately succeeding a condition state to the action state preceding the condition (Figure 4). As implied by the previous statement, the action being moved must exist in every action state immediately succeeding the condition. However, this is not the

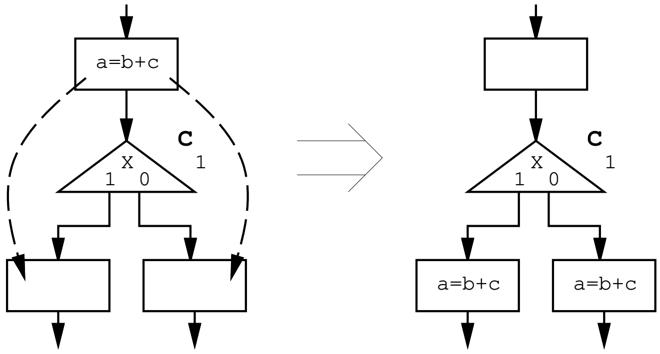

Figure 3: Action move down across a Condition state.

case when moving a subexpression, which need only exist in any of the action states immediately succeeding the condition. This transformation is similar to code hoisting in compiler optimization.

```
move_action_xcond_up(AS1, AS2, A1)
{ /* AS1 is the source state, AS2 is the destination state */
  if succ(AS2) and pred(AS1) are not the same condition state, C1, then
    abort transformation

  for each state ASi in succ(C1)
    if detect_action(A1, ASi) fail to find A1 then
      abort transformation

  if dcheck_action(A1, C1) or
     dcheck_action(A1, AS2) indicates a conflict then
    abort transformation

  copy_action(A1, AS1)
  for each state ASi in succ(C1)
    remove_action(A1, ASi)
}

move_subex_xcond_up(AS1, AS2, SE1)
{ /* AS1 is the source state, AS2 is the destination state */
  if succ(AS2) and pred(AS1) are not the same condition state, C1, then
    abort transformation

  for each state ASi in succ(C1) {
    for each Ai in detect_subex(SE1, ASi)
      factor_subex(SE1, Ai)
  }

  if SE1 was not found in any state the
    abort transformation, no subexpression to move

  if dcheck_subex(SE1, C1) or
     dcheck_action(SE1, S2) indicates a conflict then
```

```
    abort transformation

  create new action A2 for SE1
  copy_action(A2, AS1)
}
```

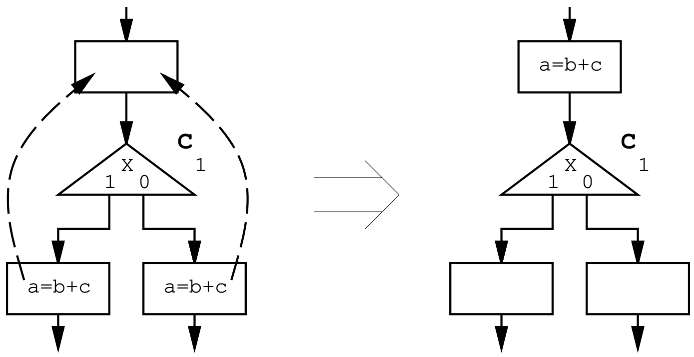

Figure 4: Action move up across a Condition state.

The transformations for action movement across join states are complementary to the ones for conditions (Figure 5). For these transformations action states must immediately precede and succeed the join state. Movement of an action or subexpression up across a join state is similar to movement down across a condition. This transformation moves an action or subexpression from the action state immediately succeeding the join state to the action states immediately preceding the join.

```
move_action_xjoin_up(AS1, AS2, A1)
{ /* AS1 is the source state, AS2 is one of the destination states */
  if pred(AS1) and succ(AS2) are not the same join state, J1, then
    abort transformation

  for each ASi in pred(J1)
    if dcheck_action(A1, ASi) indicates a conflict
      abort transformation

  for each ASi in pred(J1)
    copy_action(A1, ASi)
  remove_action(A1, AS1)
}

move_subex_xjoin_up(AS1, AS2, SE1)
{ /* AS1 is the source state, AS2 is one of the destination states */
  if pred(AS1) and succ(AS2) are not the same join state, J1, then
    abort transformation

  for each ASi in pred(J1)
    if ASi is not an action state or
       dcheck_action(A1, ASi) indicates a conflict
      abort transformation
```

```
  if detect_subex(SE1, AS1) fails to find occurrences of SE1 then
    abort transformation
  else
    for each Ai in detect_subex(SE1, AS1)
      factor_subex(SE1, Ai)

  create a new action A2 for SE1
  for each ASi in pred(J1)
    copy_action(A2, ASi)
}
```

Movement of an action down across a join state is similar to movement up across a condition (Figure 5). This transformation moves an action from the action states immediately preceding a join state to the action state succeeding the join.

```
move_action_xjoin_down(AS1, AS2, A1)
{ /* AS1 is the source state, AS2 is the destination state */
  if succ(AS1) and pred(AS2) are not the same join state, J1, then
    abort transformation

  for each ASi in pred(J1)
    if detect_action(A1, ASI) fails then
      abort transformation

  if dcheck_action(A1, AS2) indicates conflict then
    abort transformation

  copy_action(A1, AS2)
  for each ASi in pred(J1)
    remove_action(A1, ASi)
}
```

When used in conjunction, these movement transformations should enable a user to move an action to any valid point in the description without altering the designs functionality.

The last transformation we will describe deals with the merging of condition states (Figure 6). Merging conditions capitalizes on the inherent parallelism that is possible when one conditional branch immediately follows another conditional branch. In the semantics of the model we use, no arithmetic or relational operations need be performed to evaluate the condition. Condition are test for particular values of variables and will typically be implemented as sets of logic gates within the control unit. Merging conditions changes the control of the design such that the results of the two sets of condition tests are considered together in a single state rather than in two separate states. A precondition of the condition merge is that the condition states and their corresponding join states be nested with no intervening state.

```
merge_conditions(C1, C2)
{ /* C2 is assumed to be nested inside C1 */

  find join state, J1, for C1
  find join state, J2, for C2

  if succ(C1) is not C2 or
     succ(J2) is not in J1 then
    abort transformation

  merge_conds(C1, C2)
}
```

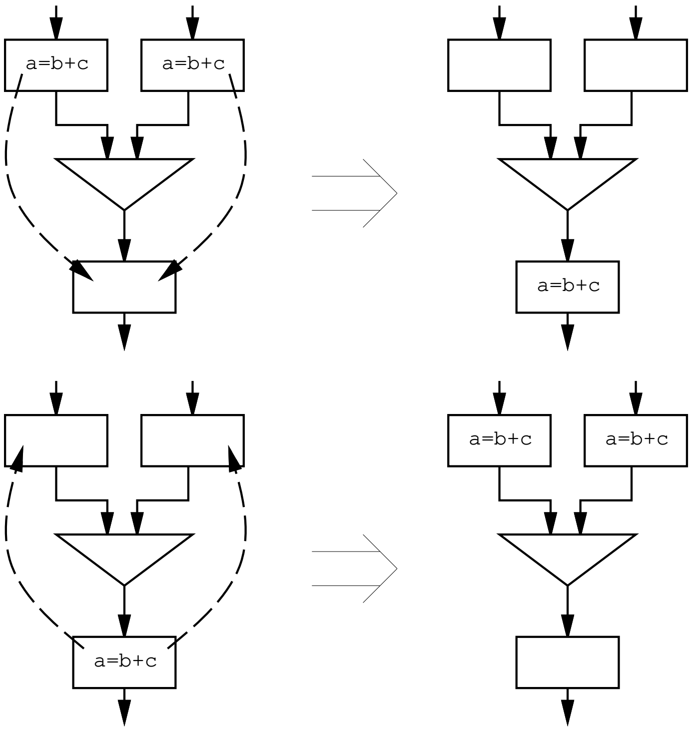

Figure 5: Action moves across a Join state.

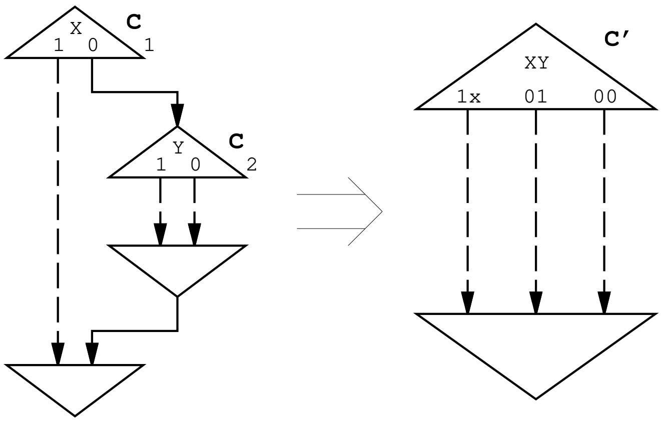

Figure 6: merging condition states $\mathbf{C}_1$ and $\mathbf{C}_2$.

6 Behavioral Correctness of Transformations

The transformations described in the previous section are only allowed when the result is free of data-dependency conflicts. In the notation used for defining a data-dependency graph, to be free of data-dependency conflicts, the previously given conditions for the transformations can be rewritten.

For the following discussions, let:

DD be the data-dependency graph for an SFSM, **SFSM**.
SFSM′ be the SFSM resulting from applying a transformation to **SFSM**.
DD′ be the data-dependency graph for **SFSM′**.

Given: v_i, v_j are instances of the same name variable.
For **SFSM′**

C1: $\nexists v_i, v_j \mid d(v_i, v_j)$ in **DD** and $s(v_i) = s(v_j)$ in **DD′** and either v_i or v_j is in the moved action.

C2: $\nexists v_i, v_j \mid v_i, v_j$ are definitions and $s(v_i) = s(v_j)$ in **DD′**

C3: $\nexists v_i, v_j$ such that v_i is a definition, v_j is a use and
$\neg d(v_i, v_j)$ in **DD** and $successor(s(v_i), s(v_j))$ in **DD′**

We will now show how the transformations preserve behavior by maintaining equivalence of data-dependency graphs.

6.1 Addition/Deletion of State

Since only empty action states are inserted or removed, these transformations do not effect how variable instances are partially ordered. If a state S' is inserted between S_1 and S_2 to create **SFSM′**, the path $S_1 \to S_2$ in **SFSM** is equivalent to the path $S_1 \to S' \to S_2$ in **SFSM′**, as far as the data-dependency graphs are concerned.

Given: state S' is inserted between S_1 and S_2, $S_1 \mapsto s_1$, $S_2 \mapsto s_2$
$successor(s_1, s_2)$ in **DD** $=$ $successor(s_1, s_2)$ in **DD′**.
if $successor(s_1, s_2)$ implies $successor(s_i, s_j)$ in **DD**,
i.e., $successor(s_i, s_1)$ and $successor(s_1, s_2)$ and $successor(s_2, s_j)$
then $successor(s_1, s_2)$ implies $successor(s_i, s_j)$ in **DD′**
else the path (if any) from s_i to s_j is the same in both **DD** and **DD′** (i.e., not through insertion path).
Therefore, $\forall s_i, s_j, successor(s_i, s_j)$ in **DD** $=$ $successor(s_i, s_j)$ in **DD′**, **(D1)** is the same in both graphs.

A similar argument can show the same equivalence for the *successor* function for a state deletion as well.

Because there are no actions in a state that is added or deleted, no additional variable instances are introduced into the data-dependency graph. Therefore, $\nexists v_i \mid v_i$ is a definition and $successor(s_i, s(v_i)), successor(s(v_i), s_j)$ for some s_i, s_j evaluates the same in **DD** and **DD′**, i.e., **(D2)** is the same in both graphs. Since **(D1)** and **(D2)** evaluate the same in both graphs, it is apparent that $d(v_i, v_j)$ evaluates the same in **DD** and **DD′** meaning that **DD** $=$ **DD′**.

6.2 Action Movement

6.2.1 Straight

To prove that moving an action into an adjacent action state under the prescribed conditions preserves a data-dependency graph, we show that contradiction of the data-dependency graph is not possible.

Assume $d(v_i, v_j)$ true in **DD** and $\neg d(v_i, v_j)$ true in **DD′**.
From the definition of the function d, one of the following must be true:

1. $successor(s(v_i), s(v_j))$ true in **DD** and $\neg successor(s(v_i), s(v_j))$ true in **DD**$'$.

2. $\exists v_k$ in **DD**$'$, but $\nexists v_k$ in **DD** $\mid v_k$ is a definition and $successor(s(v_i), s(v_k))$, $successor(s(v_k), s(v_j))$

Since only movement with an adjacent state is involved, the first condition implies that v_j was in the state after v_i, but moved into the same state as v_i.

$$successor(s(v_i), s(v_j)) \text{ in } \mathbf{DD} \Longrightarrow s(v_i) = s(v_j) \text{ in } \mathbf{DD}',$$

However, this is in direct contradiction with condition **C1** for the transformations. So this situation can not arise.

From the second condition we can imply that, in **DD**, $\neg successor(s(v_i), s(v_k))$ or $\neg successor(s(v_k), s(v_j))$ while in **DD**$'$, $successor(s(v_i), s(v_k))$ and $successor(s(v_k), s(v_j))$. Again, since only adjacent states are dealt with,

$\neg successor(s(v_i), s(v_k))$ in **DD** and $successor(s(v_i), s(v_k))$ in **DD**$'$

implies

$s(v_i) = s(v_k)$ in **DD**.

Which contradicts the definition of the data-dependency graph. Likewise,

$\neg successor(s(v_k), s(v_j))$ in **DD** and $successor(s(v_k), s(v_j))$ in **DD**$'$

implies

$s(v_k) = s(v_j)$ in **DD**.

This means $\neg d(v_k, v_j)$ is true in **DD**. But $successor(s(v_k), s(v_j))$ is true in **DD**$'$, a direct contradiction of transformation condition **C3**. So, the second or the above conditions cannot arise. This shows that a dependency in **DD** is preserved in **DD**$'$. We'll now show similarly that this transformation also does not introduce additional dependencies.

Assume $\neg d(v_i, v_j)$ true in **DD** and $d(v_i, v_j)$ true in **DD**$'$.
From the definition of the function d, one of the following must be true:

1. $\neg successor(s(v_i), s(v_j))$ in **DD** and $successor(s(v_i), s(v_j))$ true in **DD**$'$.

2. $\exists v_k$ in **DD**, but $\nexists v_k$ in **DD**$'$ $\mid v_k$ is a definition and $successor(s(v_i), s(v_k))$, $successor(s(v_k), s(v_j))$

The first situation is in direct contradiction with condition **C1** for the transformations and will not occur. Since only adjacent states are affected, from the second situation, it is implied that one of the following must is true:

$$successor(s(v_i), s(v_k)) \text{ in } \mathbf{DD} \Longrightarrow s(v_i) = s(v_k) \text{ in } \mathbf{DD}'$$
$$\text{or}$$
$$successor(s(v_k), s(v_j)) \text{ in } \mathbf{DD} \Longrightarrow s(v_k) = s(v_j) \text{ in } \mathbf{DD}',$$

If $s(v_i) = s(v_k)$ in **DD**$'$, this would be a violation of condition **C2** since v_i and v_k are both definitions. $successor(s(v_k), s(v_j))$ in **DD** $\Longrightarrow s(v_k) = s(v_j)$ in **DD**$'$ can only happen if v_k was in the action state before v_j. This means $d(v_k, v_j)$ was true in **DD**. But this contradicts condition **C1**. Therefore, we have shown this transformation will not create a dependency that did not exist in the original description.

6.2.2 Across Conditions or Joins

Unlike the "straight" movement transformation, the transformations that move actions across condition or join states also transform the data-dependency graph. We will show that the data-dependency graph resulting from such a transformation is equivalent to the original.

While these transformations are different in effect than the "straight" movement transformation, the validity of these transformations can be proved in a similar fashion. For example, when an action is moved "down" across a condition state, the transformation can be considered as a set of "straight" moves. That is, consider each path from the state preceding the condition state to a state succeeding the condition state. If it would be valid to move the action along each path, then the transformation is valid. Since each of these individual moves does not create or destroy any dependencies, the transformation as a whole also preserves the data dependencies. However, for this transformation new variable instances are created. We will show this transformation maps a dependency graph $\mathbf{DD}$ to an equivalent graph $\mathbf{DD}'$ that contains new variable instances.

For moving an action down across a condition, we copy the action into each state succeeding the condition state. To maintain equivalence between the original data-dependency graph, $\mathbf{DD}$, and the transformed graph, $\mathbf{DD}'$, the transformation also performs the following mapping:

$(v_i, v_j \in \mathbf{V})$	in **DD**		in$\mathbf{DD}'$
v_i and v_j are not in the action moved:	$d(v_i, v_j)$	$\Longrightarrow$	$d(v_i, v_j)$
v_i is in the action moved:	v_i	$\Longrightarrow$	$\{v_i^1, \ldots, v_i^n\}$
v_i is in the action moved and is a definition:	$d(v_i, v_j)$	$\Longrightarrow$	$d(v_i^x, v_j)$ where $v_i^x \in \{v_i^1, \ldots, v_i^n\}$
v_i is in the action moved and is a use:	$d(v_j, v_i)$	$\Longrightarrow$	$d(v_j, v_i^x)$ where $v_i^x \in \{v_i^1, \ldots, v_i^n\}$

Moving an action "up" across a join performs a similar mapping. Moving an action "up" across a condition or "down" across a join performs a converse mapping, where the variable instances in the duplicate actions are mapped to a new single instance. From this mapping, it is apparent that for every variable instance and dependency in $\mathbf{DD}$ there is an equivalent in $\mathbf{DD}'$, therefore these transformations maintain the equivalence $\mathbf{DD} \equiv \mathbf{DD}'$.

6.3 Condition Merging

Like the straight movement of an action, condition merging does not create new variable instances, but changes the relative order of conditional expressions. However, this transformation also changes the structure of the data-dependency graph. Given two condition states, $\mathbf{C_1}$ and $\mathbf{C_2}$, in $\mathbf{SFSM}$ the conditional expressions for these states are mapped to a condition state $\mathbf{C}'$ in $\mathbf{SFSM}'$. We will show that any dependencies associated with the variable instances in the conditional expression remain valid after the transformation.

For the following let, $s_c i$ be the node in the data-dependency graph for condition state $\mathbf{C_i}$. For a transformation that merges $\mathbf{C_1}$ and $\mathbf{C_2}$ in $\mathbf{SFSM}$ into a condition state $\mathbf{C}'$ in $\mathbf{SFSM}'$, the transformation alters the data-dependency graph as follows:

Basically, the nodes $s_c 1$ and $s_c 2$ in $\mathbf{DD}$ are mapped to s' in $\mathbf{DD}'$.

for any edge from a node s_i to $s_c 1$ in $\mathbf{DD}$, $\exists$ an edge from a node s_i to s' in $\mathbf{DD}'$
$successor(s_i, s_c 1)$ in $\mathbf{DD}$ = $successor(s_i, s')$ in $\mathbf{DD}'$

for any edge from a node s_i to $s_c 2$ in $\mathbf{DD}$, $\exists$ an edge from a node s_i to s' in $\mathbf{DD}'$
$successor(s_i, s_c 2)$ in $\mathbf{DD}$ = $successor(s_i, s')$ in $\mathbf{DD}'$

for any edge from node $s_c 1$ to a node s_i in $\mathbf{DD}$, $\exists$ an edge from s' to s_i in $\mathbf{DD}'$
$successor(s_c 1, s_i)$ in $\mathbf{DD}$ = $successor(s', s_i)$ in $\mathbf{DD}'$

for any edge from node $s_c 2$ to a node s_i in $\mathbf{DD}$, $\exists$ an edge from s' to s_i in $\mathbf{DD}'$
$successor(s_c 2, s_i)$ in $\mathbf{DD}$ = $successor(s', s_i)$ in $\mathbf{DD}'$

$\forall v_i \in \mathbf{V}$, if $s(v_i) = s_c 1$ or $s(v_i) = s_c 2$ in $\mathbf{DD}$, then $s(v_i) = s'$ in $\mathbf{DD}'$.

This preserves the ordering in the data dependency graph. For any path from s_i to s_j the graph $\mathbf{DD}$ if the path does not go through either of the transformed nodes, then $successor(s_i, s_j)$ is also true in $\mathbf{DD}'$. If the path does go through one of the transformed nodes, e.g., $s_c 1$, then $successor(s_i, s_j)$ implies $successor(s_i, s_c 1), successor(s_c 1, s_j)$. But from the above transformations, this maps to $successor(s_i, s'), successor(s', s_j)$ which means $successor(s_i, s_j)$ is true in $\mathbf{DD}'$.

7 Status and Future Work

The behavior-preserving transformations described in this paper have been implemented in C on Sun workstations, and are currently working in the BIF design environment [DuHG90] [HaGa91]. A graphical user interface, XBIF [DuCH91], allows the designer to interactively select and apply these transformations to a scheduled behavior represented as an BIF Operations-Based State Table (OBST). This is an attractive interface for applying these types of transformations, since the designer can view the scheduled behavior in a form that preserves the original HDL flavor, yet provides a clear picture of the state sequencing and conditions. An attempt to apply a transformation incorrectly is aborted, and the user is notified via a message on the screen that the original behavior may not be preserved.

To complement the rescheduling transformations, we intend to enhance the XBIF environment to display design metrics similar to those described in [BrGa90] and [WRJF91]. Inclusion of component allocation information will be necessary to derive design characteristics such as the number and type of functional units needed, utilization of registers and functional units, estimated area of design, estimated time length of states, etc. Such information will give the designer direct feedback on the impact of the rescheduling transformations on the implementation that may be synthesized from the specification Initially, component allocation information will be generated by automatic tools. But potentially, the environment can be enhanced to allow simultaneous manipulation of a design's scheduling and component allocation.

8 Acknowledgments

This research was supported in part by NSF grant MIP-9009239 and SRC contract 91-05-146. We are grateful for their support.

9 Summary

In this paper, we first formally defined the original behavior from an imperative HDL as represented by a well-structured CDFG. We formally defined the SFSM, a well-structured, abstract FSM that is appropriate for capturing the behavior after scheduling in high-level synthesis. We proved the equivalence of the scheduled CDFG and the SFSM by describing construction rules to go between these design models. We explained the advantages of maintaining both design models. Furthermore, we showed that the SFSM (and hence the scheduled CDFG) is a subset of the general FSMD design model used in high-level synthesis. We then outlined primitive transformations and user-applicable behavior preserving transformations for interactive rescheduling in high-level synthesis. Since Automatic schedulers may use specific (and sometimes simplified) design models, there is a need to provide a manual rescheduling capability for users. This permits designers to incorporate additional knowledge, such as floorplanning and layout information, early in the design process. Our current work attempts to find good metrics to help designers apply these transformations effectively.

10 References

References

[AhSU86] Aho, A., Sethi, R., Ullman, J. *Compilers: Principles, Techniques and Tools*, Addison-Wesley, 1986.

[BhLe90] Bhasker, J. and Lee, H. "An Optimizer for Hardware Synthesis," *IEEE Design and Test of Computers*, pp. 20–36, October 1990.

[BrGa90] Brewer, F. and Gajski, D., "Chippe: A System for Constraint Driven Behavioral Synthesis," *IEEE Transactions on Computer-Aided Design*, Vol. 9, no. 7, pp. 681–695, July 1990.

[Camp90] Camposano, R., "Behavior-preserving Transformations for High-Level Synthesis," *Proceedings of the MSI Workshop on Hardware Specification, Verification, and Synthesis: Mathematical Aspects*, pp. 106–128, Springer Verlag, 1990.

[CaRo89] Camposano, R. and Rosenstiel, W., "Synthesizing Circuits from Behavioral Descriptions", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Vol. 8, no. 2, pp. 171–180, February 1989.

[CaTa89] Camposano, R. and Tabet R. "Design Representation for the Synthesis of Behavioral VHDL Models," in J.A. Darringer, F.J. Rammig, Editors, *Proceedings of the* 9^{th} *International Symposium on Computer Hardware Description Languages and their Applications*, 1989.

[DeJo91] DeJong, G. "Data Flow Graphs: System Specification with the Most Unrestricted Semantics," *Proceedings of the European Design Automation Conference*, 1991.

[DuCH91] Dutt, N., Cho, J. and Hadley, T. "A User Interface for VHDL Behavioral Modeling," *Proceedings of the* 10^{th} *International Symposium on Computer Hardware Description Languages and their Applications*, 1991.

[DuHG90] Dutt, N., Hadley, T. and Gajski, D., "An Intermediate Representation for Behavioral Synthesis," *Proceedings of the* 27^{th} *Design Automation Conference*, pp. 14–19, 1990.

[GDWL92] Gajski, D., Dutt, N., Wu, A., Lin, S., *High-Level Synthesis: Introduction to Chip and System Design*, Kluwer, 1992.

[HaGa91] Hadley, T. and Gajski, D., "A Decision Support Environment for Behavioral Synthesis", Technical Report 91-17, University of California at Irvine, Aug. 1991.

[Keut87] Keutzer, K., "DAGON: Technology Binding and Logic Optimization by DAG Matching," *Proceeding of the* 24^{th} *Design Automation Conference*, pp. 341–347, 1987.

[LeWo91] Leeser, M. and Wolf, W., "Preserving Design Behavior without Register-Transfer Equivalence", *High Level Synthesis Workshop*, 1991.

[MSBS91] Malik, S., Sentovich, E., Brayton, R., Sangiovanni-Vincentelli, A., "Retiming and Resynthesis: Optimizing Sequential Networks with Combinational Techniques", IEEE Transactions on CAD, Vol. 10, No. 1, pp. 74–84, Jan. 1991.

[Marw86] Marwedel, P., "A New Synthesis Algorithm for the MIMOLA Software System" *Proceedings of the* 23^{rd} *Design Automation Conference*, 1986.

[McFa78] McFarland, M., "The Value Trace: A Data Base for Automated Digital Design", Technical Report DRC-01-04-80, Carnegie Mellon University, Dec. 1978.

[OrGa86] Orailogulu, A. and Gajski, D., "Flow Graph Representation," *Proceedings of the* 23^{rd} *Design Automation Conference*, pp. 503–509, 1986.

[PoRa91] Potkonjak, M. and Rabaey, J., "Optimizing Resource Utilization using Transformations", *Proceedings of the IEEE Conference on Computer-Aided Design*, pp. 88–91, 1991. 1991.

[Rose86] Rosenstiel, W., "Optimizations in High Level Synthesis," *Microprocessing and Microprogramming (18)*, pp. 543–549, 1986.

[RuGa90] Rundensteiner, E. and Gajski, D., "A Design Representation for High-Level Synthesis", Technical Report 90-27, UCI, Sep. 1990.

[Walk82] Walker, R., "A Transformation Package for the Behavioral Level of the CMU-DA System", Master's Project Report DRC-01-14-82, Carnegie Mellon University, Dec. 1982.

[WaTh91] Walker, R. and Thomas, D., "Behavioral Transformation for Algorithmic Level IC Design", IEEE Transactions on CAD, Vol. 8, No. 10, pp. 1115-1128, Oct. 1989.

[WRJF91] Walker, R., Ramabadran, S., Joshi, R., and Flatland, S., "Increasing User Interaction During High-Level Synthesis," 24^{th} *International Symposium on Microarchitecture*, pp. 1–10, Dec. 1991.