Analysis and Synthesis of Concurrent Digital Circuits Using Control-Flow Expressions

Claudionor Nunes Coelho, Jr. and Giovanni De Micheli, *Fellow, IEEE*

***Abstract*—In this paper, we present a novel *modeling style* and *control synthesis* technique for system-level specifications that are better described as a set of concurrent descriptions, their synchronizations, and constraints. The proposed synthesis procedure considers the degrees of freedom introduced by the concurrent models and by the environment in order to satisfy the design constraints. Synthesis is divided in two phases. In the first phase, the original specification is translated into an algebraic system, for which complex control-flow constraints and quantifiers of the design are introduced. In the second phase, we translate the algebraic formulation into a finite-state representation, and we derive an optimal control-unit implementation for each individual concurrent part. In the implementation of the controllers from the finite-state representation, we use flexible objective functions, which allow designers to better control the goals of the synthesis tool, and thus incorporate as much as possible their knowledge about the environment and the design.**

I. Introduction

THE USE of synthesis tools has gained great acceptance in industry. Three of the reasons for its success are the increasing complexity of the circuits, the need for reducing time to market, and the need to design circuits optimally. In order to meet the tight requirements of today's marketplace, designers have to rely on the specification at higher levels of abstraction, and in particular, rely on models that describe the specification at a level higher than the logic level and register-transfer level (RTL) [1].

In these designs specified at higher levels, the system to be synthesized is usually modeled as a set of sequential components consisting of operations and their dependencies, e.g., as in the case of a dataflow. We call *process* each sequential component. Processes have been successfully synthesized at chip-level by experimental and/or commercial high-level synthesis tools. The synthesis task in these tools involved the scheduling of operations over a discrete time and the binding of these operations to components.

This paper considers systems that are better described as a set of synchronous concurrent processes and their synchronization, which we call here a *multiprocess* or *system-level* design. Although attempts to use high-level synthesis tools to synthesize multiprocess descriptions have been made, these techniques are usually not well suited for system-level designs for three reasons. First, most high-level synthesis tools synthesize one process at a time, thus not considering some degrees of freedom in the optimization. Second, the model used for specifying and handling the interface in most high-level synthesis is very simple, and does not easily support modifications. Finally, standard cost functions used in high-level synthesis are simple, i.e., the goal of the synthesis tool is usually the minimization of area or delay. In system-level designs, we may have to quantify not only area and delay, but more complex cost measures, such as bus or microprocessor utilization.

The use of single process techniques in the synthesis of multiprocess descriptions imposes severe limitations on the implementations; in some cases it even prevents valid implementations from being found. Multiprocess descriptions require the use of more complex algorithms and techniques other than the ones used for single process synthesis. These complex techniques involve the utilization of the degrees of freedom of the other processes during the synthesis of a single process, the use of synchronization among processes to further optimize the synthesis tasks, the modification of the control-flow over time, as required by the specification, and the selection of the different goals of the synthesis tools.

Multiprocess descriptions also require specifications of complex constraints. For example, when synthesizing single process models, the tool does not have to consider the synchronization among concurrent descriptions. However, when synthesizing multiprocess descriptions, the interrelations among different processes must be considered. In addition, interrelations of the different parts in an interface do not need to be static. For example, a synchronous RAM has different requirements in terms of cycles for the different modes of operation. The ability of adding complex timing constraints results in a greater flexibility with respect to a specification.

1.1. Research Objective

In this paper, we present a formal model to analyze control-flow intensive synchronous system-level specifications (operating under a single clock), and a methodology to synthesize control-units for the concurrent parts of the design. In this methodology, the control-flow of the description is first abstracted into an algebraic system, here called *control-flow expressions*, manipulated, and then translated into its state space, where the control-unit is synthesized. Our technique

Manuscript received September 20, 1994; revised December 6, 1995 and April 16, 1996. This work was supported by ARPA under Grant DABT 63-95-C-0049 9 115 432. The work of C. N. Coelho was supported by Scholarship 200 212/90.7 from CNPq/Brazil, and by a fellowship from Fujitsu Laboratories of America. This paper was recommended by Associate Editor M. McFarland.

C. N. Coelho, Jr. is with the Departamento de Ciência da Computação, ICEx/UFMG, Belo Horizonte, MG, Brazil.

G. De Micheli is with the Computer Systems Laboratory, Stanford University, Stanford, CA 94305 USA.

Publisher Item Identifier S 0278-0070(96)05724-7.

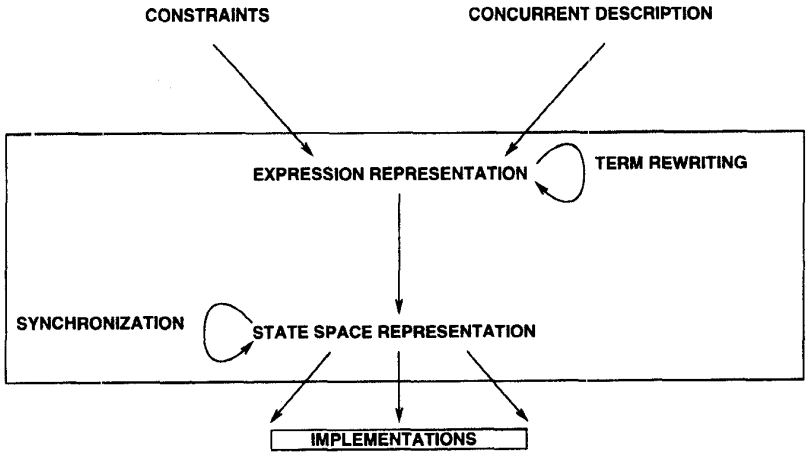

Fig. 1. Design flow for synthesizing multiprocess descriptions.

also extends previous synthesis approaches because it considers processes with arbitrary control-flow. We emphasize that our system utilizes both representations during synthesis, i.e., an algebraic representation and a state space representation, and that these two representations are necessary for the efficient manipulation of the specification for different types of transformations. Fig. 1 presents a pictorial view of our design flow. We assume that the design is originally specified by some hardware description language (HDL), such as VHDL, Verilog HDL, or HardwareC, and compiled into some control-dataflow graph (CDFG) model. We assume that the compilation from the HDL into the CDFG model is a direct mapping, and that control-flow expressions can be obtained from the CDFG through abstraction. So, in all examples in this paper, we will use the original HDL specification instead of the CDFG representation. It should be emphasized here that we focus neither on a specific CDFG model nor on an HDL language representation, but on a modeling style for concurrent synchronous systems and a synthesis technique for their controllers.

The constraints of the system are manually entered in the tool from some constraint language that includes synchronization, timing, and binding constraints. For example, the specification of synchronization constraints is already present in the Esterel [2] language. The HardwareC language allows the specification of timing and binding constraints that are used by the synthesis tool. These constraints will guide the synthesis tool during the synthesis of the control-unit implementations.

In the next section, we present some examples of where our formulation can be used and how those problems can be solved. In the following section, we define the algebra of *control-flow expressions*, its axioms, the representation of the design space, and a comparison with existing formalisms. In Section IV, we show how constraints can be represented in *control-flow expressions*, and how we restrict the solution space with respect to the constraints. In Section V, we show how the algebra of control-flow expressions can be transformed into a finite-state representation. In Section VI, we present our synthesis method using an 0–1 integer linear programming specification with Boolean constraints. In Section VII, we present some applications of this methodology with implementation results, followed by some conclusions.

II. Motivation

This section presents examples of real designs that either cannot be synthesized or are synthesized suboptimally by usual high-level synthesis tools. We show intuitively that optimal and valid implementations can be obtained only if synchronization, dynamic binding, and dynamic scheduling are considered during the design space exploration. Then, in the rest of the paper, we present formal methods to obtain optimal solutions to these synthesis problems.

One of the major problems of using current high-level synthesis tools to synthesize system-level designs is that the synthesis tool must consider how the environment affects the whole system. Since the specification of the environment in which the circuit is going to execute is generally a formidable task, the user must have a better control over the synthesis tool in order to obtain optimal results. The user can interact with our synthesis tool by specifying complex constraints and flexible cost functions. The necessity of this interaction will become apparent in the next examples.

2.1. Synchronization Synthesis and Dynamic Binding

2.1.1. Ethernet Coprocessor: In this example, we show how we can synchronize multiple processes to share the same critical resource. This synchronization is synthesized by considering the degrees of freedom among the different processes that share the critical resource. Dynamic binding is achieved by allowing several processes to instantiate the same resource at different times. In this example, a constraint that crosses process boundaries exists, i.e., the critical resource should not be used by more than one process at a time.

The block diagram of Fig. 2 is the block diagram of an ethernet coprocessor. This coprocessor contains three units: an execution unit, a reception unit, and a transmission unit. These three units are modeled by thirteen concurrent processes, with three processes accessing the bus: *DMAxmit, DMArcvd*, and *enqueue*. The problem we want to solve is the synthesis of the

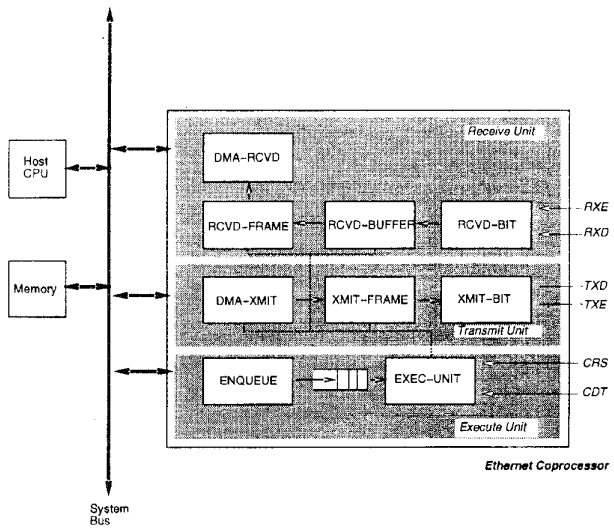

Fig. 2. Ethernet coprocessor.

synchronization among the three processes such that any bus access for the three processes is free of conflicts. Note that the difficulty in solving this problem comes from the transfers that are nondeterministic over time, i.e., *we do not know a priori when each process accesses the bus*, since this operation is control dependent. Also, the transfers of different processes are uncorrelated, i.e., knowing that one process accesses the bus at a specific time does not imply the transfers in other processes are known.

This problem has been solved for the simplified assumption that the processes are dataflows executing at the same rate [3]. Note that in the problem described here, however, we do not know when each bus access will take place, since there may be loops and conditionals in the specification that will make the bus accesses execute at different rates. Thus, the approach described in [3] cannot be used. Filo *et al.* [4] addressed the problem of rescheduling transfers inside a single loop or conditional to reduce the number of synchronizations among processes. This method is restrictive because all transfers that are optimized must be enclosed in the same loop or conditional, and only the synchronization due to the transfers is considered during the simplification. A synchronization is eliminated if there are two transfers that are executed sequentially or in parallel and the synchronization of the first one is correlated to the second transfer. As we are going to show later, our formalism allows processes to be specified by their control-flow with an abstraction on the dataflow parts, and thus will subsume the solutions found by both of these procedures. Also, our formalism achieves the simplification of synchronization that crosses loops and conditionals, and we do not restrict this simplification to only correlated transfers in the specification.

Let us first consider an abstraction of the original specification that captures only the bus accesses. Furthermore, in order to be able to discuss this problem throughout this paper, we will assume a set of reduced behaviors for *DMArcvd*, *DMAxmit*, and *enqueue* such that the resulting behavior is small enough that can be easily understood. Fig. 3 presents the behaviors we assume for these descriptions in this paper, in a pseudöVerilog code. In this figure, the constructs that do not belong to the language, such as `write bus`, are represented in typewriter style; reserved words of Verilog are represented in bold; and other legal syntactic constructs are represented in italics.

Note that the processes are control-dominated specifications where the flow of control is modified by some set of wait statements. In this example, also, note that the priority of *enqueue* should be the smallest one, since the execution of the bus access in this process may be delayed. On the other hand, if the bus accesses of the other processes are delayed, the controller will not be able to deliver data at the interface at the proper rate.

If we assume that every operation takes one clock cycle, an implementation for the synchronization mechanism of the bus should establish a temporal relation between *enqueue* and the two other processes *DMAxmit* and *DMArcvd*. This temporal relation should include any data dependent operation of the two other processes, such as the conditional *transmission ready*, and it should also consider when the other processes access the bus. A possible solution to this problem would be

```
module enqueue;
always
begin
   wait (posedge clock);
   while (transmission ready)
   begin
      wait (posedge clock);
      wait (posedge clock);
   end
   read bus;
end
endmodule
```

In this implementation, we have to wait the first cycle because *DMArcvd* is accessing the bus in the first cycle. During the second cycle, *enqueue* will be able to access the bus only if *DMAxmit* is not accessing it. In the following cycle, however, *DMArcvd* will be accessing the bus again, and *enqueue* will have to wait for another cycle. We will show later how this controller could be obtained automatically for the process *enqueue*.

2.1.2. Protocol Conversion: In this section, we show how we can use synchronization synthesis in order to synthesize the controller for converting the PCI bus protocol [5] into a synchronous DRAM protocol. In particular, we will provide here the conversion between reading and writing cycles of a PCI bus into synchronous DRAM cycles. Fig. 4 shows the diagram of a computer using a PCI bus, and a synchronous DRAM (SDRAM) memory bank. Both protocols can use single or burst mode transfers, with the difference that SDRAM's burst mode are limited to at most eight transfers on the same row that are one cycle apart from each other.

Informally, a PCI bus cycle begins with an address phase, followed by one or more data phases. Wait states can be inserted in the data phase by either the microprocessor or by

```
module DMArcvd;
always
begin
   write bus;
   data = receive(from_xmit_frame);
end
endmodule
```

```
module DMAxmit;
always
begin
   initialize variables
   wait (transmission ready);
   read bus;
end
endmodule
```

```
module enqueue;
always
begin
   wait (free bus);
   read bus;
end
endmodule
```

Fig. 3. Abstracted behaviors for *DMArcvd, DMAxmit* and *enqueue.*

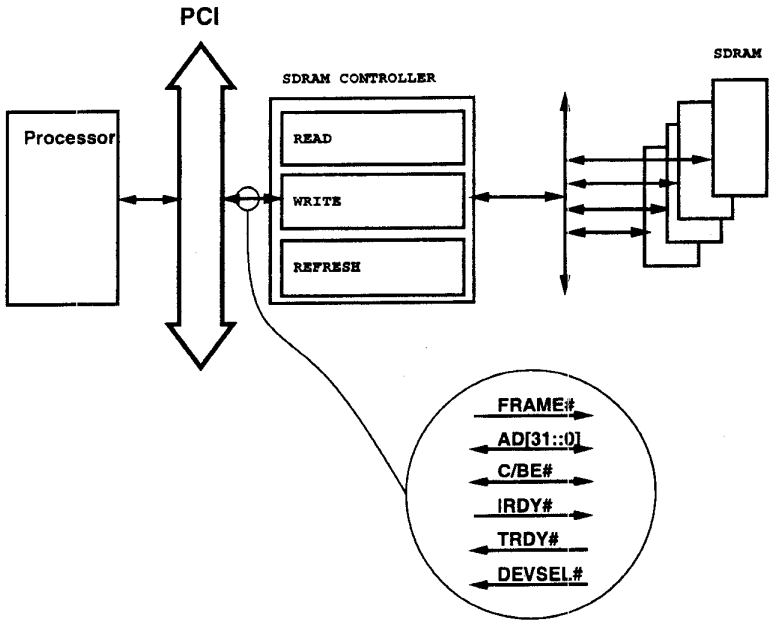

Fig. 4. Protocol conversion for PCI bus computer.

the memory. For burst mode transactions, we assume here a linear increment of the address space.

The synchronous DRAM reading protocol begins by a row address selection (RAS) phase followed by a column address selection (CAS) phase. After the CAS phase and a fixed number of cycles, the SDRAM will produce data at a rate of one word/cycle.

During the generation of the protocol converter, a control-unit implementation is selected to combine the behaviors of both SDRAM and PCI bus protocols. Implementations satisfying these protocol conversion constraints were obtained in the system described in [6]. In our approach, we will show how such constraints can be combined with timing and resource binding constraints in order to generate optimal controllers.

2.2. Dynamic Scheduling

In this example, we show how we can specialize a design by incorporating dynamic scheduling constraints from an interface. Splitting the interface specification from the design specification was addressed in [7]–[10]. One of the main advantages of abstracting interface implementation details at the higher levels of abstraction is that more degrees of freedom can be explored during synthesis.

In such techniques, the transfers among processes are abstracted in terms of communication operations (such as a *send* operation). During synthesis, the best protocol and communication medium is selected to implement a particular transfer. The selection and synthesis of the protocol interface will impose complex scheduling constraints to the design, as we will see below.

Consider a system that has an ASIC and an embedded processor, such as the one given in Fig. 5. Assume the ASIC communicates with the microprocessor either through a synchronous memory or through a synchronous FIFO. For example, this structure has been used in hardware-software codesign [11], [12]. In this system, the transfers to the memory and to the FIFO are determined at run-time by the proper selection of the address. The interface timing is also determined at run-time, since the timing specifications for these two components are different, as given in Fig. 6. In essence, a data transfer may take either one or three cycles to complete. Thus, the timing constraint specification should also reflect the mismatch between the timing of the components.

The specification of interface constraints has been used in the past by Nestor [10], Ku [8], and Borriello [7]. They used min/max scheduling constraints to annotate the design specification. The use of these constraints, however, is limited

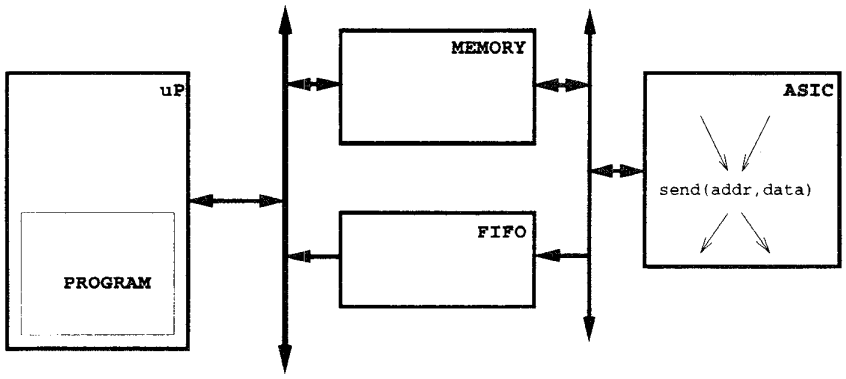

Fig. 5. System architecture.

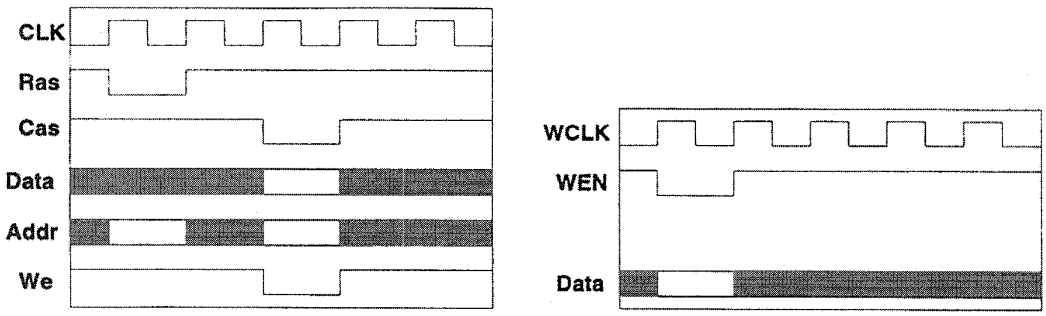

Fig. 6. Writing cycles for (a) synchronous DRAM and for (b) synchronous FIFO.

to static constraints. In the example presented above, the specification of the interface requires the design to contain implementation details, which is not desirable for the reasons given previously.

Assuming that the address selection for the memory module is called s, the constraint that we need to specify is a three-cycle operation or a one-cycle operation, depending on s. Thus, the interface can no longer be specified in terms of fixed minimum/maximum delay between operations, since the execution time of the operation is dependent on the address selection. In order to synthesize the protocol for the *send* operation given above, we must consider a dynamic schedule for this operation.

This can be achieved by using the alternative composition in the constraint specification. For example, one possible representation for this constraint could be

$$\begin{array}{l} synchronize\ with\ \text{“}send\text{”}\ operation \\ \textbf{if}\ (s) \\ \quad delay\ for\ \text{“}send\text{”}\ is\ 3\ cycles \\ \textbf{else} \\ \quad delay\ for\ \text{“}send\text{”}\ is\ 1\ cycle. \end{array}$$

We will show that using the algebra of control-flow expressions, we can represent this constraint as the following compact representation

$$s : Ras \cdot 0 \cdot \{Cas, data\} + \overline{s} : data$$

where *Ras* is an abstraction to the RAS cycle of the RAM, *Cas* is an abstraction of the CAS cycle of the RAM, 0 is a one-cycle delay operation, *data* is an abstraction of the data transfer, and $\overline{s}$ means that s is *false*.

During the synthesis procedure, the *send* operation is bound to an implementation that observes this constraint. In this case, the implementation is exactly the control that waits either one or three cycles, depending on s.

In this example, the two different communications mechanisms assume different possible behaviors for the environment. Depending on how the environment requires data, one mode should be highlighted over the other for some transfer by the proper selection of an objective function.

III. Control-Flow Expressions

This section presents the definition of the algebra of *control-flow expressions*, which is a formal model for representing the control-flow in system-level designs. As the name suggests, *control-flow expressions* are used for the analysis of the control-flow of the design, by abstracting away the dataflow details.

3.1. Abstraction from the Original Specification

We consider in this paper system-level designs that will be synthesized as synchronous circuits running under the same clock. In the synthesis of these designs, we need to represent

the interactions among the concurrent parts, which can be best modeled at the control-flow level.

We assume in our computation model that the specification will be partitioned in terms of a control-flow and a dataflow, as described in [1], [13], [14]. In this model, variables, their operations, and operands are placed in the dataflow, and the language constructs of the specification language are placed in the control-flow. In addition, we assume that any I/O operation between a process and the process external environment will be placed in the dataflow.

In this model, the control-flow and dataflow will communicate through events. The control-flow will generate output events to the dataflow that will sensitize the execution of operations in the dataflow. The dataflow will generate input events to the control-flow that will trigger the different execution paths.

Example 1: In Fig. 7, we show how the control-flow portion of a description can be abstracted in terms of the events it generates. The control-flow of the specification generates output events a_1, a_2, a_3, and a_4. Event a_1, for example, triggers the execution of the path in the dataflow that will complement dx. We represent the dataflow by an implementation in terms of a datapath for illustrative purposes only. In general, we do not assume any particular dataflow implementation, since control-flow will be able to encode several possible datapath implementations.

The datapath of Fig. 7 generates input events c_1 and c_2 that will trigger the execution of the loop and the execution of the alternative path, respectively.

The reader should note that the control-flow does not make any assumptions on the possible values of its input events over time. In this example, we assume that entering the loop (when even c_1 is generated) and exiting the loop are equally probable. □

3.2. Algebra of Control-Flow Expressions

The algebra of control-flow expressions is defined by the abstraction of the specification in terms of the sensitization of paths in the dataflow, and by the compositions that are used among these operations. As presented in the previous section, we view the communication between the dataflow and control-flow as an event generation/consumption process. More formally, we call the output events generated from the control-flow *actions* (from some alphabet $\mathcal{A}$). We assume that each action will execute in one unit of time (or cycle). If an operation executes in multiple cycles, they will be handled by a composition of single-cycle actions.

Example 2: We abstract the computation $x = y * z$ of some HDL by action a, which then substitutes all occurrences of this computation in the specification. □

We represent the input events of a control-flow by *conditionals*, which are symbols from an alphabet $\mathcal{C}$. The conditionals in a control-flow expression will enable different blocks of the specification to execute. Guards will be defined as the set of the Boolean formulas over the set of conditionals.

Definition 3.1: A guard is a Boolean formula on the alphabet of conditionals. We will use $\mathcal{G}$ to denote the set of guards over conditionals.

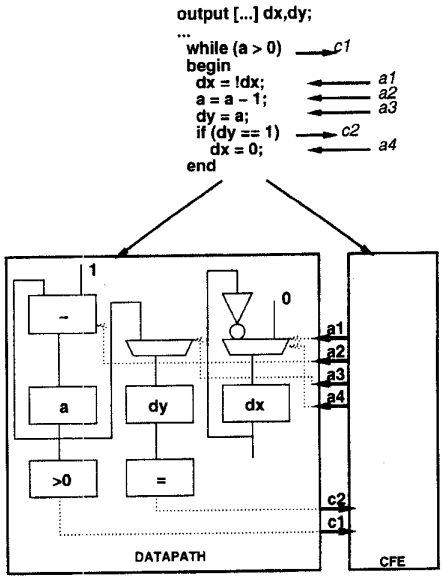

Fig. 7. Partitioning of specification into control-flow/dataflow.

We assume that each guard and conditional is evaluated in zero time. At the end of this section, we compare the assumptions on the execution time of actions, conditionals, and guards with the synchrony hypothesis.

Example 3: In the specification *if* $(x \geq y)x = y * z$, a conditional c abstracts the binary relational computation $x \geq y$. If at some instant of time, the *guard* c is *true*, $x = y * z$ is executed. If at some instant of time, the *guard* $\neg c$ is *true*, the else branch (which is null in this case) is executed. □

As discussed in the introduction, we assume systems modeled by a set of operations, dependencies, concurrency, and synchronization. We encapsulate subbehaviors of this system in terms of processes, which are represented by control-flow expressions and correspond to an HDL model. In our representation, each process has a label from some alphabet $\mathcal{F}$ to control-flow expressions.

We define the set Σ as the alphabet of actions, conditionals and processes $\Sigma = \mathcal{A} \cup \mathcal{C} \cup \mathcal{F}$.

The compositions that are defined in the algebra of control-flow expressions are the compositions supported by existing HDL's. Verilog HDL, for example, supports sequential composition, alternative composition, loops, forks, and unconditional repetition. The same set of compositions is also supported in VHDL and HardwareC, and thus is supported by control-flow expressions. Since alternative compositions and loops in these languages are guarded, their corresponding compositions in CFE's will also be guarded.

We define the set $\mathcal{O} = \{sequential(\cdot),\ alternative(+),\ guard(:), loop(*), infinite(\omega),\ parallel(||)\}$ as being the valid compositions of control-flow expressions. The formal definition of the algebra of control-flow expressions is presented below

Definition 3.2: Let $(\Sigma, \mathcal{O}, \delta, \epsilon)$ be the algebra of control-flow expressions where:

Σ is an alphabet that is subdivided into the alphabet of actions, conditionals, and processes;

$\mathcal{O}$ is the set of composition operators that define sequential, alternative, guard, loop, infinite, and parallel behavior;
δ is the identity operator for alternative composition;
ϵ is the identity operator for sequential composition.

For the sake of simplicity, we restrict the sets of behaviors definable in control-flow expressions in the following way: it should always be possible to obtain a control-flow expression without any process variables. In this sense, the set of process variables have the same cardinality as the set of control-flow expressions without process variables. In this paper, whenever we refer to a CFE p, we are referring to the CFE defined by the process variable p.

We consider a special action called 0, which corresponds to a no-operation or abstraction of the computation. Action 0 executes in one unit-delay (just as any other action), but it corresponds either to an unobservable operation of a process with no side effects or to a unit-delay between two computations.

In Definition 3.2, we introduced the symbol δ that is called here *deadlock*.[1] The symbol δ is defined as $\delta \triangleq false : p$, where p is any control-flow expression. The deadlock symbol is an identity for alternative composition. This means that the branch of the alternative composition represented by the deadlock is never reachable. Later, we show that these branches can in fact be removed.

We also introduced the symbol ϵ, which is called here the *null computation*. The *null computation* symbol is defined as a computation that takes zero time. For example, this symbol can be used to denote an empty branch of a conditional. This symbol behaves as the identity symbol for sequential composition.

The semantics of the major control-flow constructs in HDL are related to control-flow expressions in the table in Fig. 8, where p and q are processes ($p, q \in \mathcal{F}$) and c is a conditional ($c \in \mathcal{C}$). In this figure, we relate CFE to the control-flow structure of Verilog HDL [15]. In this paper, we assume that guards (:) have precedence over all other composition operators; loops and infinite composition $(*, \omega)$ have precedence over the remaining compositions; sequential composition $(\cdot)$ has precedence over alternative and parallel composition; alternative composition $(+)$ has precedence over the parallel composition. In addition, we use parentheses to overrule this precedence and for ease of understanding. Although it is not necessary, we will at times replace parentheses by square brackets for clarity.

Informally, we define the behavior of the compositional operators of CFE's as follows: the sequential composition of two processes p and q means that q is executed only after p is executed. The parallel composition means that both p and q begin execution at the same time, and any operation following $p||q$ will begin execution when both p and q have completed. Note that the parallel composition does not assume that p and q must terminate at the same time. The alternative composition means that a deterministic choice is first made with respect to c and $\neg c$ to decide whether the CFE p or q is executed, respectively. The loop composition means that p is executed while the guard c is *true*. The infinite composition means that p is executed infinitely many times upon reset.

[1] Deadlock was the name given to δ in process algebras. In synthesis, δ denotes code that is unreachable due to synchronization. Since its properties are the same as the properties for deadlock in process algebras, we used the latter name, for the sake of uniformity.

Composition	HL Representation	CF Expression
Sequential	**begin** p; q **end**	$p.q$
Parallel	**fork** p; q **join**	$p\|\|q$
Alternative	**if** (c) p ; **else** q ;	$c : p + \overline{c} : q$
Loop	**while** (c) p ;	$(c : p)^*$
	wait $(!c)$ p ;	$(c : 0)^*.p$
Infinite	**always** p ;	p^ω

Fig. 8. Link between Verilog HDL constructs and control-flow expressions.

Note that in our definition of the syntax of CFE's, every loop and every alternative branch is guarded by ":", which makes the different branches of alternative and loops distinct. This restricts the specification of loop bodies and alternative branches to only accept deterministic choices with respect to the guards.

We will use the following shorthand notation for control-flow expressions. The control-flow expression p^n will denote n instances of p composed sequentially

$$\underbrace{(p \cdot \cdots \cdot p)}_{n}$$

which corresponds, for example, to a counting loop that repeats n times in some HDL. The control-flow expression $(x : p)^{<n}$ will denote a control-flow expression in which at most $n-1$ repetitions of p may occur. This CFE is equivalent to $(x : p + \overline{x} : \epsilon)^{n-1}$.

In our original specification, we assumed that every action in $\mathcal{A}$ takes a unit-time delay in CFE's, and that every guard takes zero time delay. Then, we could possibly design a system where after choosing a particular branch of an alternative composition (e.g., after choosing c is *true* in $c : p + \overline{c} : q$) and executing the first action of process p, the execution of this action would make $\overline{c}$ *true* and thus also enabling the execution of q. In order to avoid this erroneous behavior, we adopt a weaker version of the *synchrony hypothesis* [16].

Assumption 3.1: Let p be a process and c be a guard that guards the execution of p (defined as $c : p$). Any action of p is assumed to execute after c has been evaluated to true. In other words, $c : p$ can be viewed as $(c : \epsilon) \cdot p$. First, the conditional is evaluated to true, then the process p that is guarded by c is executed, and other assignments to c will possibly affect future choices only.

3.3. Axioms of CFE's

In this section, we present the axioms for the algebra of control-flow expressions. These axioms provide the theoretical

TABLE I
AXIOMS OF CONTROL-FLOW EXPRESSIONS

$c_1 : p + c_2 : q$	$=$	$c_2 : q + c_1 : p$	*(+ is commutative)*
$(c_1 : p + c_2 : q) + c_3 : r$	$=$	$c_1 : p + (c_2 : q + c_3 : r)$	*(+ is associative)*
	$=$	$c_1 : p + c_2 : q + c_3 : r$	
$(c_1 : p + c_2 : q) \cdot r$	$=$	$c_1 : p \cdot r + c_2 : q \cdot r$	*(· distributes to the left with +)*
$(p \cdot q) \cdot r$	$=$	$p \cdot (q \cdot r)$	*(· is associative)*
	$=$	$p \cdot q \cdot r$	
$c_1 : p + c_1 : p$	$=$	$c_1 : p$	*(+ is idempotent)*
$1 : p$	$=$	p	
$0 : p$	$=$	δ	
$c_1 : p + \delta$	$=$	$c_1 : p$	*(δ is the identity element for +)*
$\delta \cdot p$	$=$	δ	*(δ is the zero element for ·)*
$p \cdot \epsilon$	$=$	p	*(ϵ is the identity element for ·)*
$\epsilon \cdot p$	$=$	p	
$c_1 : c_2 : p$	$=$	$(c_1 \wedge c_2) : p$	
$a \| \| b$	$=$	$(a \cup b)$	if $a \cup b$ synchronize
$a \| \| b$	$=$	δ	if $a \cup b$ does not synchronize
$a \| \| b$	$=$	$b \| \| a$	
$a \| \| 0$	$=$	a	
$a \| \| \epsilon$	$=$	a	
$a \cdot p \| \| b \cdot q$	$=$	$(a \| \| b) \cdot (p \| \| q)$	
$a \cdot p \| \| b$	$=$	$(a \| \| b) \cdot p$	
$(c_1 : p + c_2 : q) \| \| r$	$=$	$c_1 : (p \| \| r) + c_2 : (q \| \| r)$	

background that will be used to build the finite-state machine representation for control-flow expressions in Section V.

The algebra of control-flow expressions inherits its formalism from a subset of process algebras [17] that is suitable for describing synchronous systems, called the *algebra of regular synchronous processes*. We further extend this algebra by specifying Boolean variables as guards of processes. The following proposition holds for CFE's.

Proposition 3.1: CFE's are a subset of regular synchronous process algebras.

In Table I, we present the axioms of control-flow expressions that are derived from the axioms of the algebra of synchronous processes, where $a, b \in \mathcal{M}^{\mathcal{A}}$ (the set of multisets of actions), $p, q, r \in \mathcal{F}$ (processes) and $c_1, c_2, c_3 \in \mathcal{G}$ (guards).

The alternative composition has δ as its identity component. It is commutative, and associates to the right or left. The sequential composition has ϵ as its identity component. It associates to both the right and left, and it is only distributive to the left with respect to the alternative composition. This implies that $p \cdot (c_1 : r + c_2 : s) \neq c_1 : p \cdot r + c_2 : p \cdot s$. The intuitive meaning for $p \cdot (c_1 : r + c_2 : s)$ being different from $c_1 : p \cdot r + c_2 : p \cdot s$ is that we abstracted away the computation of p, c_1 and c_2, and thus we cannot answer the question on whether action p affects the choice of c_1 or c_2, or if the environment needs some value from p for making a decision on whether c_1 or c_2 should be true. If we assumed this transformation were valid, we could make the decision for all branches of the specification on the start by propagating the guards toward the beginning.

On the other hand, if we assumed that $p \cdot (c_1 : r + c_2 : s)$ were equivalent to $p \cdot c_1 : r + p \cdot c_2 : s$, we would be in fact assuming that system were noncausal (its current choices depending on the future value of conditionals), and in this case, we could also have propagated all those decisions to the initial start time of the system modeled by the CFE.

The parallel composition assumes a *synchronous execution semantics*, also known as maximal parallelism semantics. In this execution semantics, if two processes are executed in parallel, then one action of each process is executed atomically at the same time. We represent the actions that execute together by multisets of actions. For example, if multiset a defines $\{a_1, \cdots, a_n\}$, where each $a_i \in \mathcal{A}$, actions $a_1, \cdots, a_n$ are executed at the same time. The set consisting of multisets of actions is represented here by the symbol $\mathcal{M}^{\mathcal{A}}$. If two multisets $a = \{a_1, \cdots, a_n\}$ and $b = \{b_1, \cdots, b_m\}$ are composed in parallel, the resulting multiset $\{a_1, \cdots, a_n, b_1, \cdots, b_m\}$ is represented by $a \cup b$. We sometimes abuse our notation for multisets and use a_i for $\{a_i\}$ if it can be inferred by the context that a_i represents the multiset $\{a_i\}$.

In the definition of the axioms of CFE's, we showed that the result of the parallel composition of two multisets a and b is dependent on some synchronization between a and b. Although a formal definition of synchronization will be presented in the next section, we will give an informal definition that will allow the reader to understand its meaning.

Processes synchronize in control-flow expressions by defining multisets of actions that always have to execute at the

same time, and by defining multisets of actions that should never execute at the same time.

Loops and infinite computations can be defined by control-flow expressions with process variables. The loop composition $(c:p)^*$ is equivalent to recursive process $q = c:p\cdot q + \overline{c}:\epsilon$, where p is a process variable. The infinite composition p^ω is equivalent to the recursive process $q = p\cdot q$. Their axioms can be determined by applying those equations into axioms of the original algebra.

3.4. Comparison of Control-Flow Expressions with Existing Formalisms

Control-flow expressions are very useful as a modeling and abstraction formalism for CDFG's, since the translation from a CDFG into CFE's is straightforward. In this section, we compare CFE's with other formalisms that were used to model the control-flow, while abstracting the dataflow information: regular expressions, path expressions, finite-state machines, Petri-nets, algebra of concurrent processes (ACP), calculus of communicating systems (CCS), timing expressions, and BFSM's, although this list is by no means exhaustive.

- The algebra of **regular expressions** [18] is used represent strings accepted/emitted by a finite-state machine. This algebra is represented by $(\Sigma, +, \cdot, *)$, where Σ is the alphabet of characters accepted/emitted, $+$ denotes alternative composition, $\cdot$ denotes sequential composition, and $*$ denotes zero or more repetitions of a subexpression.

  Regular expressions have been used in the modeling of the control-flow of sequential programs [19], [20]. In order to specify the control-flow in terms of an input/output behavior, regular expressions must be extended to guard alternative branches and loops. Also, in the case of parallel descriptions, a parallel operator must be added. However, this parallel operator is redundant for regular expressions, since the left and right distributivity of the sequential operator with respect to the alternative operator allow concurrency to be traded by nondeterminism [21]. Such expressiveness does not exist in control-flow expressions, because the sequential operator does not distribute to the right with respect to the alternative operator.

  CFE's also extend regular expressions by defining infinite behaviors, which could be achieved only by extending regular expressions to ω-regular expressions [22].
- **Path expressions** [23] are equivalent to regular expressions, with the addition of parallelism. However, instead of a synchronous execution semantics for the parallel composition, path expressions assume an interleaved execution semantics. CFE's also extend path expressions by providing guards to alternative branches and loops, in the same way CFE's extended regular expressions.
- A **finite-state machine** [18] recognizer is a tuple $(\Sigma, S, \delta, S_0, F)$, where Σ is a set of inputs, S is the set of states, $\delta : S\times\Sigma \rightarrow S$ is the transition function, S_0 is the set of initial states, and F is the set of final states. In the case of finite-state machines as computational engines, we also define an output alphabet O, and either the output transition function $\Delta : S \rightarrow O$ (in the case of a Moore machine) or $\Delta : S\times\Sigma \rightarrow O$ (in the case of a Mealy machine). Parallelism in finite-state machines is defined only at the transition level, in which several outputs may be generated at the same time. At this level, however, the duration for each output has already been determined, and any transformation of the specification that modifies this execution time cannot be performed.

  A specification consisting of a set of concurrently executing finite-state machines can also be considered in this model, as in the case of reactive system languages, such as StateCharts [24] and SDL [25]. In these languages, the system is modeled as a set of hierarchical concurrent finite-state machines, and the system's state is defined to be the state of the Cartesian product of all concurrently executing finite-state machines. As in the case described in the previous paragraph, at the level of finite-state machines, the execution time for the operations has already been decided, and thus any transformation that changes the execution time of operations cannot be performed, without requiring a restructuring of the finite-state machine.
- **Petri-nets** [26] are represented by the tuple (T, P, δ, I), where T is the set of transitions, P is the set of places, and $\delta \subseteq T\times P \cup P\times T$ defines the transition relation (or firing) from transitions to places and vice-versa. A marking in Petri-nets is an assignment of natural numbers (tokens) to places. I is the initial marking of the Petri-net.

  A state in a Petri-net is a marking of places. Transitions between states are achieved by having a marking that becomes another marking by firing some transition. This firing occurs when one transition of the net has all incoming places with more than one token. The transition takes one such token from each incoming place and puts one additional token in every outgoing place. Since only one firing can occur at any time, this model can only represent interleaved concurrent systems.

  One possible extension of Petri-nets is the synchronous firing semantics [27]. In this semantics, the set of firings that can occur at the same time is specified along with the Petri-net. Similarly to the concurrent finite-state machine model, any transformations that changes the execution time of the operations, or the structure of the graph cannot be easily performed.
- **Process algebra** [17] and **CCS** [28] correspond to a family of representations used to formally model concurrent systems. In these models, we view the system as a set of operations that are represented by *actions*, and their compositions in terms of sequential composition, nondeterministic choice, parallel composition, and communication. Concurrency usually refers to interleaved concurrency, which is represented by nondeterministic choice; and synchronous concurrency is defined in terms of communication.

  These representations can be considered as a superset of control-flow expressions. If we restrict the set of specifiable behaviors to regular and synchronous processes, then control-flow expressions will have the same representation capabilities of process algebras and CCS.

One of the unique features of control-flow expressions that was defined previously in this paper is that we distinguish actions from conditionals. This allows the system to capture the reactive nature of hardware systems better, and as a result, control-flow expressions will fit the model used for synthesis better.

- **Timing expressions** [29], [30] is a model for describing behaviors of sequential systems and specifying sequential constraints a sequential system has to satisfy [29]. In timing expressions, the sequential system is represented by expressions that may take different values over time. When compared to control-flow expressions, we see that timing expressions will be better suited to represent the control information at lower levels of descriptions, whereas control-flow expressions will be better suited for representing the control-flow at higher-levels of descriptions. In addition, control-flow expressions can be considered as a superset of timing expressions, since CFE's can be used to represent systems containing hierarchical series-parallel specifications, whereas in timing expressions parallelism can occur only at the highest level.
- **BFSM's** [31] are a generalization of finite-state machines with partial timing information on the relative execution time of the states. Through synthesis, a complete time (or schedule) is obtained. This model closely resembles the algebra of control-flow expressions because it was used for modeling and synthesis of control-dominated specifications. However, the lack of a synchronization formalism and the lack of a formal model for constraint specification—which is restricted to scheduling constraints—prevents BFSM's from being used in more complex problems. As opposed to CFE's, which uses both expression and finite-state machine representations for a concurrent system, the translation from the specification to a finite-state machine description is performed too early with BFSM's, and thus, optimizations that would be best used at the expression level—such as hierarchical abstraction and rewriting—would not be available to the synthesis process. Finally, BFSM is a model best suited for representing the control-flow of languages in which parallelism is specified at the process level, such as VHDL. If used to represent the control-flow of languages that can specify series-parallel composition of systems, such as Verilog HDL, its representation and constraint specification becomes cumbersome.

When compared to the formalisms presented above, control-flow expressions are able to capture more succinctly the control-flow information, abstraction from the original specification, and the degrees of freedom. When considering specifications in terms of CDFG's (or in terms of the corresponding HDL code) control-flow expressions fit perfectly as a modeling tool of the control behavior for synthesis of system-level specifications.

IV. Constraint Specification

In the previous section, we presented the algebra of control-flow expressions, and how to abstract the dataflow information and represent the control-flow of the design. Real designs consist of specification and design constraints. In this section, we show how to use CFE's to represent constraints, such as scheduling, binding, and synchronization.

The specification of a system at higher levels of abstraction requires the modeling of nondeterminism, since at these levels, not all synthesis decisions have been made. In the algebra of control-flow expressions, we model these nondeterministic choices of the design by guarding the choices with decision variables, which quantify the design space.

In this section, we present the incorporation of design constraints by control-flow expressions. Both the specification and the constraints will be converted to a finite-state representation in the next section, where we will be able to obtain the controllers satisfying design constraints.

4.1. Quantification of the Design Space

We represent here the design space and constraints by means of *decision variables*, which are used as guards of CFE's.

Definition 4.1: A decision variable d is a variable guarding the execution of a control-flow expression whose value is determined by the synthesis procedure. Its possible values are defined as the set of Boolean formulas over some set $\mathcal{D}$.

A decision variable is a Boolean variable that quantifies a constraint, i.e., whenever the decision variable is true, the constraint is satisfied. A simple implementation that has been sought in the past is the assignment of decision variables to constant values over time [32]. Later, we show how to obtain assignments to the decision variables that considers the "state" of the system being synthesized. Thus, in some cases, the set $\mathcal{D}$ will be the set of conditionals $\mathcal{C}$, with the Boolean constants $\{0, 1\}$. When we obtain a finite-state machine satisfying the constraints in the next section, the different machines from which we can choose will be uniquely determined by different assignments to the decision variables.

In the algebra of control-flow expressions, we are going to use decision variables as guards of expressions, so we will need to extend guards to allow decision variables and conditionals to be composed together. Because decision variables will uniquely determine the satisfaction of a constraint, we only need to compose guards with conjunctions of decision variables or their complements. This also states that any nondeterminism from the specification will be uniquely guarded by a Boolean guard.

Definition 4.2: A guard is a conjunction of decision variables (or their complements) and a Boolean formula over the set of conditionals.

Example 4: Consider the code $w = y * z; u = w + 3$. Assume both the multiplication and the addition take one clock cycle, and that $w = y * z$ is represented by action a and $u = w + 3$ is represented by action b. A *constraint* between a and b, or the quantification of all possible schedules such that b occurs after a is represented by the CFE $a \cdot (x : 0)^* \cdot b$, where $a, b \in \mathcal{A}$, and $x \in \mathcal{D}$. In this CFE, the possible schedules are quantified by the different assignments of the decision variable x over time.

Possible assignments could be

$$\begin{aligned} & a \cdot b \\ & a \cdot 0 \cdot b \\ & a \cdot 0 \cdot 0 \cdot b \\ & \vdots \\ & a \cdot 0 \cdot \cdots \cdot 0 \cdot b. \end{aligned}$$

The first assignment corresponds to an assignment of x to *false* after the execution of action a. The second assignment corresponds to an assignment of x to *true* after the execution of a, then to *false*. The other assignments have a similar correspondence. □

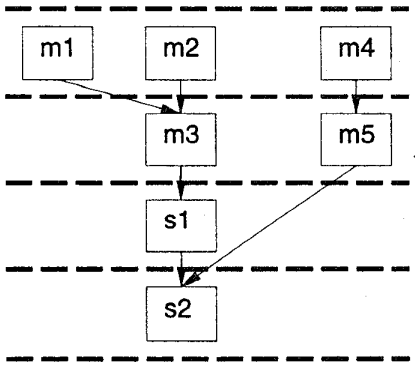

Fig. 9. CDFG of a differential equation example.

4.2. Constraint Representation

Constraints are properties that any implementation needs to satisfy. We consider here a subset of constraints that can be specified as scheduling constraints, binding constraints and synchronization constraints. More complex specifications can be achieved by composing these constraints using control-flow expressions.

Timing constraints will be defined in terms of control-flow expressions. In binding constraints, we will use expression rewriting, i.e., the incorporation of binding constraints as a modification of the original CFE. Both timing and binding constraints will use decision variables as quantifiers of the design space. Finally, synchronization constraints will use multisets of actions that should occur at the same time and multisets of actions that should never occur at the same time.

The constraints will be defined in terms of the actions that appear in a control-flow expression, which we define below as the *support* of a CFE.

Definition 4.3: The support of a control-flow expression p is defined as the set of actions that are executed in p.

Example 5: The support of a CFE $p = (a \cdot b)^{\omega} || (c \cdot d \cdot e)^{\omega}$, written as S_p, is the set of actions of p. Here, $S_p = \{a, b, c, d, e\}$. □

Each action defined in the support of a CFE will have a shadow action, which executes every time the corresponding action executes.

Definition 4.4: A shadow of an action a, written as σ_a, is defined to be an action that does not correspond to any operation of the original specification and executes every time action a is executed.

Example 6: In the CFE $(a \cdot b \cdot c)^{\omega}, \sigma_a$ is executed every time a is executed, σ_b is executed every time b is executed, and σ_c is executed every time c is executed. □

4.2.1. Scheduling Constraints: Scheduling constraints are constraints that specify the timing relations among computations. Although we will only define minimum and maximum timing constraints here, we can specify and handle a much richer set of constraints with control-flow expressions, including loops, alternative composition and synchronization, as opposed to the constraints that are handled in other CAD tools, such as [7], [8], [33], [34]. The specification of scheduling constraints using control-flow expressions can be also considered as an extension of path constraints defined by [33].

Let us assume p to be a CFE representing a specification of a design with support S_p. Suppose we want to represent initially simple minimum and maximum constraints between two actions a and b, with $a, b \in S_p$.

Definition 4.5: A minimum timing constraint of n cycles between two actions a and b, whose shadow actions are σ_a and σ_b, can be represented by the CFE $(x : 0)^* \cdot \sigma_a \cdot 0^{n-1} \cdot (y : 0)^* \cdot \sigma_b$, where x and y are decision variables.

Definition 4.6: A maximum timing constraint on n cycles between two actions a and b, whose shadow actions are σ_a and σ_b, can be represented by the CFE $(x : 0)^* \cdot \sigma_a \cdot (y : 0)^{<n} \cdot \sigma_b$, where x and y are decision variables.

Let p be a control-flow expression representing a specification and let $m_1, \cdots, m_n$ be a set of CFE's representing scheduling constraints. The control-flow expression $p||m_1|| \cdots ||m_n$ will denote the application of the n scheduling constraints to the specification p.

Example 7: The design in Fig. 9 is the control-data flow graph of a subset of the loop of a differential equation solver [1]. Assume that the CFE for the specification is p, and that we want to specify a maximum timing constraint of three cycles between m_4 and s_2, which can be represented by the CFE $(x : 0)^* \cdot \sigma_{m_4} \cdot (y : 0)^{<3} \cdot \sigma_{s_2}$, where x and y are decision variables.

The application of this constraint to the CFE p is represented by a new CFE $p||(x : 0)^* \cdot \sigma_{m_4} \cdot (y : 0)^{<3} \cdot \sigma_{s_2}$. □

In the previous example, we specified conventional minimum and maximum timing constraints. As we pointed out before, CFE's can be used to specify a much broader set of scheduling constraints, and even hide interface information from the original specification, as shown in the following example.

Example 8: Let us examine the specification of the scheduling constraint presented in Section II-2. In this example, the different actions that are involved in the transmission of the data are the actions "Ras," "Cas," and "data." Associated with the action "send," we have the shadow action σ_{send}. The constraint that specifies that the send operation should take either three or one cycle(s), depending on the address selection, can be represented by the control-flow expression $(x : 0)^* \cdot (s_a : \{\sigma_{\text{send}}, Ras\} \cdot 0 \cdot \{Cas, data\} + \overline{s_a} : \{\sigma_{\text{send}}, data\})$. □

4.2.2. Binding Constraints: Binding constraints specify the possible implementations for each computation that is repre-

sented by an action. We represent binding constraints as a rewriting of the original control-flow expression.

Definition 4.7: Let p be a control-flow expression with support S_p. A rewriting of p, written as $\mathcal{R}(p)[a \leftarrow q]$, where q is a control-flow expression, is defined as the substitution of every occurrence of $a \in S_p$ in p by q.

Example 9: Assume we make the rewriting of a by $(c_1 : a_0 \cdot a_1 + c_2 : a_0 \cdot a_1 \cdot a_2)$ into $p = (a \cdot b)^\omega || (c \cdot d \cdot e)^\omega$. Then

$$\mathcal{R}(p)[a \leftarrow (c_1 : a_0 \cdot a_1 + c_2 : a_0 \cdot a_1 \cdot a_2)] = ((c_1 : a_0 \cdot a_1 + c_2 : a_0 \cdot a_1 \cdot a_2) \cdot b)^\omega || (c \cdot d \cdot e)^\omega.$$

Definition 4.8: Let p be a CFE of a specification and assume some action a can be implemented by a set of components $\{C_1, C_2, \cdots, C_m\}$. This binding constraint is represented by the CFE

$$\mathcal{R}(p)\left[a \leftarrow \sum_{1 \le i \le m} x_i : C_i\right]$$

where $\Sigma_{1 \le i \le m} x_i : C_i$ represents the alternative composition of the m terms $(x_i : C_i)$, and $x_1, \cdots, x_m$ are m decision variables.

In this expression rewriting, whenever x_i is *true*, component C_i implements the computation abstracted by action a. Note that since decision variables are assumed to take values from the set of Boolean formulas over $\mathcal{D}$, and not just the values zero or one, we may have an implementation in which some x_i enables component C_i at some time, and at a later time x_j $(i \neq j)$ enables component C_j, thus implementing dynamic binding of components.

Example 10: In this example, assume that actions $m_i, i = 1, \cdots, 5$ of Fig. 9 can be implemented by one of three multipliers M_1, M_2, M_3. Then, for the CFE p that represents this CDFG, we define the binding for each m_i as

$$\mathcal{R}(p)[m_i \leftarrow (x_{i1} : M_1 + x_{i2} : M_2 + x_{i3} : M_3)]$$

where i ranges over one to five and x_{i1}, x_{i2} and x_{i3} are decision variables. □

Note that in this section, we are only specifying binding constraints. When an assignment to the decision variables is obtained in such a way that different bindings are selected at different times, then we refer to this as dynamic binding.

4.2.3. Synchronization Constraints: Synchronization constraints specify actions that should be executed at the same time and actions that should never be executed at the same time. The former type of synchronization corresponds to the specification data transfers, or control transfer from one specification to another. The latter kind of synchronization allows one to specify exclusive use of a resource by some individual process.

We define below *ALWAYS* and *NEVER* sets, which are sets consisting of multisets of actions.

Definition 4.9: Let ALWAYS be a set consisting of multisets of actions that contains multiset X. If two actions a and b belong to the same multiset X, then a and b must always execute at the same time.

Definition 4.10: Let NEVER be a set consisting of multisets of actions that contains multiset X. If two actions a and b belong to the same multiset X, then a and b must never execute at the same time.

Example 11: Let us consider the synchronization synthesis problem presented in Section II-1-1. In this problem, let us assume the following control-flow expressions for the processes *DMArcvd, DMAxmit*, and *enqueue*, respectively

$$p_1 = [a \cdot 0]^\omega$$
$$p_2 = [0 \cdot (c : 0)^* \cdot a]^\omega$$
$$p_3 = [(x : 0)^* \cdot a]^\omega$$

where a corresponds to the bus access and zero hides the internal computation from the original specification. The conditional c hides the evaluation of *transmission ready* predicate and the decision variable x quantifies the predicate *free bus*. In this case, since we have the additional restriction that no two bus accesses should occur at the same time, we have $NEVER = \{\{a, a\}\}$.

In summary, we showed how to represent scheduling, binding, and synchronization constraints in this section. More complex constraint specifications can use these three types of constraints as building blocks, with the compositions of control-flow expressions as a way to combine these constraints.

V. Finite-State Representation

This section shows how to generate a finite-state representation from control-flow expressions. As we have shown in Fig. 1, we use both the algebraic and the finite-state representations in our synthesis tool. The algebraic representation presented in the previous sections allows us to manipulate and rewrite the expressions algebraically. The finite-state representation allows us to analyze and to synthesize the controllers for the specification.

We obtain a finite-state representation from a control-flow expression by computing all the suffixes of the expression. Informally, a suffix of a control-flow expression represents the state of the system after an n-cycle simulation of the system. We show that this state can be represented by another CFE, and we call this simulation of the CFE to obtain its suffixes a *derivative*, because of the its resemblance to the work of Brzozowski [35] who first defined derivatives of regular expressions.

In the following example, we will present the key ideas of this section in obtaining a finite-state representation for a control-flow expression by enumerating its suffixes. The algorithm will be formalized later.

Example 12: For the control-flow expression $p = (a \cdot b \cdot c)^\omega$, we wish to obtain a finite-state Mealy machine. By inspecting p, and assuming that a, b, and c are the outputs to the finite-state machine representing p, we know that a Mealy machine starting at some initial state q_0, makes a transition to some state q_1 with output a being generated. From state q_1, the finite-state machine makes a transition to some state q_2 with output b. Finally, a transition q_2 occurs to the original state q_0 with output c. The Mealy machine for this control-flow expression is presented in Fig. 10.

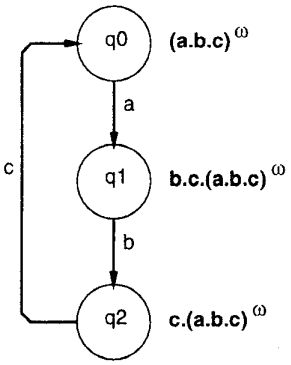

Fig. 10. Mealy machine for control-flow expression $(a \cdot b \cdot c)^\omega$.

If we now look at the possible suffixes of p, the CFE $b \cdot c \cdot (a \cdot b \cdot c)^\omega$ is obtained after simulating $(a \cdot b \cdot c)^\omega$ for one cycle, and the CFE $c \cdot (a \cdot b \cdot c)^\omega$ is obtained after simulating $b \cdot c \cdot (a \cdot b \cdot c)^\omega$ for one cycle. Thus, we can associate the states q_0, q_1, and q_2 with the suffixes $(a \cdot b \cdot c)^\omega, b \cdot c \cdot (a \cdot b \cdot c)^\omega$ and $c \cdot (a \cdot b \cdot c)^\omega$, respectively. □

What we need to show now is how to compute the suffixes of a control-flow expression, that there is only a finite number of suffixes for a given CFE, and that there is an equivalence relation between the suffixes of a control-flow expression and the states of its corresponding Mealy automaton. This is described formally in Appendix A. We suggest to the reader who is interested in the mathematical foundation of this paper to go to this appendix before proceeding to the next section.

5.1. Constructing the Finite-State Representation

In this section, we present a procedure to obtain the finite-state Mealy machine from a control-flow expression using derivatives. This Mealy machine is formally represented by $M = (I, O, Q, \delta, \lambda, q_0)$,[2] where I is the set of input variables of M, O is the set of output symbols of M, Q is the set of states, q_0 is the initial state, δ is the transition function of M, i.e., $\delta = Q \times 2^I \rightarrow Q$, and λ is the output function of M, i.e., $\lambda : Q \times 2^I \rightarrow 2^O$.

This Mealy machine is related to the set of derivatives of p in the following way. The set of input variables of M corresponds to the set of conditional and decision variables of p. The set of outputs of M corresponds to the multiset of actions of p. With each irredundant suffix s of p, we associate a state $q_s \in Q$. In particular, q_0 corresponds to the state q_p, i.e., to the CFE p itself.

The transition function (δ) and the output function (λ) are related to the CFE p in the following way. Let s be an irredundant suffix of a control-flow expression p, for which we are building the finite-state machine representation. The triple $(\gamma, \mu, \pi) \in \mathcal{G} \times \mathcal{M}^A \times \mathcal{F}$ (defined formally in Appendix A), obtained from a CFE p, indicates that the actions μ are executed when γ is true, followed by the execution of π. Assume that $(\gamma, \mu, \pi) \in \partial s$, where ∂s denotes the derivative of s. Thus, $\delta(q_s, \gamma) = \pi$ and $\lambda(q_s, \gamma) = \mu$ in M.

[2]We use the Greek letter δ to denote the transition function as used in literature. This δ is different from the δ introduced in Section III-3, but the reader should be able to easily recognize when we are referring to the *deadlock* symbol and when we are referring to the *transition function* of the Mealy machine M.

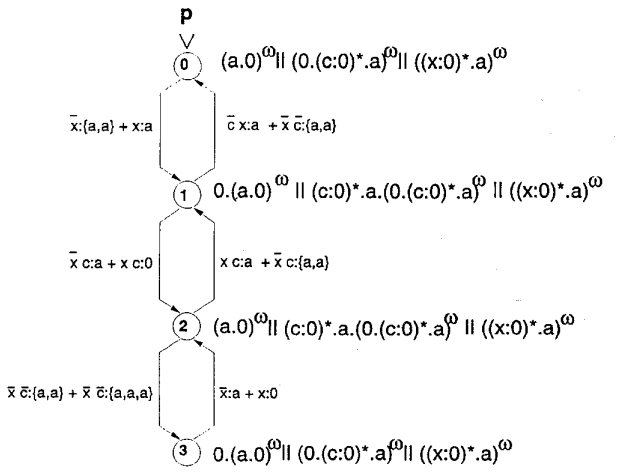

Fig. 11. Finite-state representation for synchronization synthesis problem.

Example 13: Fig. 11 shows the finite-state representation for the synchronization example whose control-flow expression was presented in Example 11 $(p_1||p_2||p_3)$. □

Note that the derivative computation does not take into account the synchronization constraints. Thus, we will need the following definitions.

Definition 5.1: A transition $\delta(q, f)$ of a finite-state Mealy machine representation of the control-flow expression p is valid if

- $\forall x \in ALWAYS, (\lambda(q, f) \cap x \neq \emptyset) \Rightarrow (x \subset \lambda(q, f))$
- $\forall x \in NEVER, x \notin \lambda(q, f)$.

The definition above states that if at least a certain action in a transition is included in some multiset of actions of the *ALWAYS* set, then all actions in this multiset should be executed in the transition. Furthermore, this transition should not include any multiset of actions of the *NEVER* set. This condition guarantees that the transition will not violate the synchronization requirements of the design.

Since some of the transitions of the Mealy machine may be invalid, we have also to check whether a state of the machine is reachable by valid transitions or not.

Proposition 5.1: The initial state q_p of the finite-state Mealy machine representing the control-flow expression p is reachable, and so is any other state $q \in Q$ such that there is at least one valid transition from another reachable state to q.

The algorithm of Fig. 12 is used to compute the finite-state Mealy machine M of a specification. The algorithm works by traversing the finite-state machine in a breadth-first search manner, and eliminating the invalid transitions and the unreachable states. The finite-state machine obtained contains only the reachable states and valid transitions of the system. The design space represented by the scheduling and binding constraints are embedded into the original control-flow expression of the specification.

Example 14: If we apply the $NEVER = \{a, a\}$ constraint to the finite-state representation of $p_1||p_2||p_3$ (shown

```
/* Breath First Search of state space represented by CFE p */
procedure Construct_FSM
{
   input:  cfe, ALWAYS, NEVER
   output: finite-state machine M
   fifo.init (cfe)                                   /* initialize fifo with initial cfe */
   while (fifo ≠ ∅) {
      cfe = fifo.first()                             /* get cfe on top of fifo */
      mark(cfe)                                      /* mark cfe as traversed and make it a state */
      derivative = ∂ (cfe)                           /* compute all cfe's one cycle apart */
      ∀(γ, μ, π) : (G × M^A × F) ∈ derivative {

                                                     /* check if it violates ALWAYS and NEVER sets */
         if (μ ∩ ALWAYS ≠ ∅)
            if (ALWAYS ⊄ μ) continue
         if (μ ∩ NEVER ≠ ∅)
            if (NEVER ∈ μ) continue
         add edge (cfe, γ : μ, π) to finite-state machine   /* transition is valid*/
         if unmarked (π)                             /* if suffix isn't a state, insert it in fifo */
            fifo.insert (π)
      }
   }
   remove unreachable states
}
```

Fig. 12. Algorithm to construct finite-state representation.

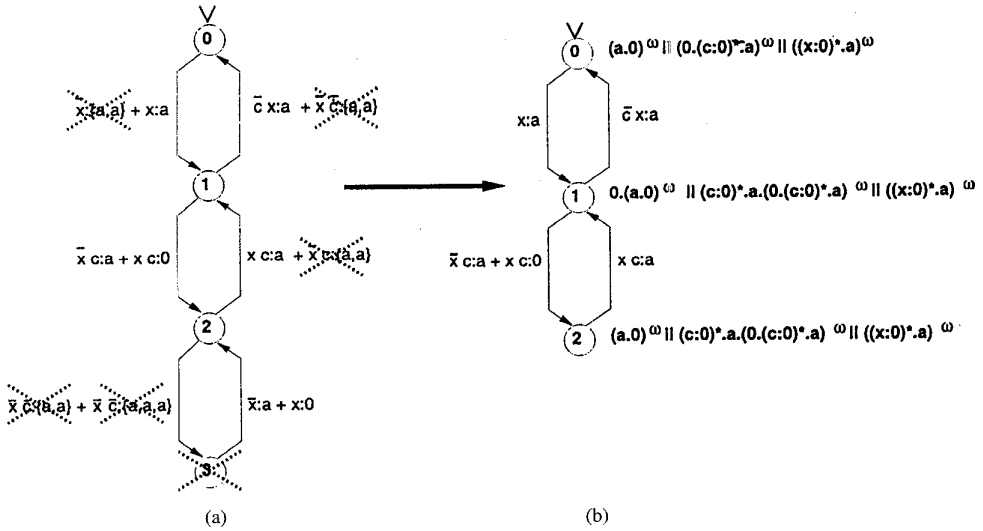

Fig. 13. Finite-state representation observing synchronization constraints. (a) Finite-state machine without applying always and never sets. (b) Finite-state machine.

in Example 13), we obtain the finite-state representation of Fig. 13(b).

Note that state 3 becomes unreachable from the initial state, and thus can be eliminated from the final finite-state representation. □

5.2. Feasibility of Solutions

In the design process, the user may want at some point to determine if there exists an implementation for the specification in the presence of a set of design constraints. The following theorem shows how one can test whether a problem is overconstrained or not.

Theorem 5.1: Suppose p is a control-flow expression along with the synchronization constraints specified by the sets ALWAYS and NEVER. If the procedure Construct_FSM (p, ALWAYS, NEVER) returns an empty finite-state machine, then the specification is overconstrained.

Proof: We know that at least one state should exist in the finite-state machine: the state corresponding to $q = p||m_1||\cdots||m_n$. If this initial state does not exist in the finite-state machine, it means that it was first generated (before the **while** loop of the algorithm in Fig. 12), but later removed from the finite-state machine because the state was unreachable. Since invalid transitions are eliminated when they violate synchronization constraints, q was overconstrained. ■

Note that the converse may not be true, however. If the overconstrained part of the specification is not large enough to make all states unreachable, then an implementation is still

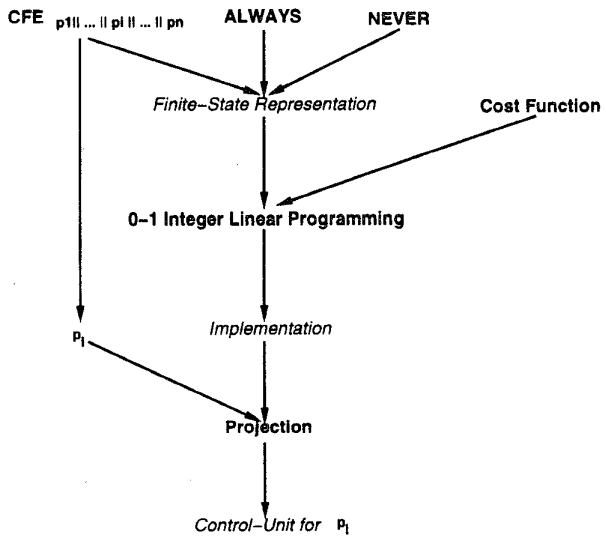

Fig. 14. Methodology for synthesizing control-units.

obtained for the parts of the specification that satisfies the constraints.

VI. Synthesizing Control-Units from the Finite-State Representation

We present in this section a methodology to synthesize control-units from the finite-state representation. Fig. 14 gives a pictorial view of the synthesis method. From the specification represented by the set of concurrent processes $p_1||\cdots||p_i||\cdots||p_n$, and the synchronization constraints expressed by the *ALWAYS* and *NEVER* sets, we obtained a finite-state machine representation by the algorithm shown in the previous section. From this finite-state representation, which already contains all feasible behaviors, we look for a feasible implementation that has been optimized with respect to a cost function. In particular, we obtain in this section the implementation by casting the synthesis problem as a 0–1 Integer Linear Programming instance. Note that the optimized finite-state representation models the system as a whole. Thus, to derive the controller for each individual process p_i, we project the set of decisions made for the implementation into p_i. This methodology can be used to synthesize the controllers of concurrent systems with arbitrary control-flows, as well as systems with environment and synchronization constraints.

The major difference between our formulation and previous approaches to synthesis, such as [32], [36]–[38] is that we do not have the notion of a control-step as a linear order over time, because of loops, synchronization, and concurrency. Whereas the control flows only in one direction in single-source single-sink dataflows, loops make the analysis of the control-flow depend on the different assignments to the conditionals. Concurrency implies that different instances of the same piece of computation require different decisions. Finally, synchronization implies that the different parts of the specification should not be treated separately. Thus, the complexity of the synthesis task becomes much higher.

The dependency of the flow of control on the conditionals and on the design constraints prevents us from formulating the synthesis problem in terms of control-steps. However, we can define the synthesis problem in terms of an equivalent entity: the state of a finite-state machine.

We will consider, thus, the finite-state machine $M = (I, O, Q, \delta, \lambda, q_0)$ defined in Section V-1 that represents the control-flow expression p and the synchronization constraints. This finite-state machine was obtained by the algorithm given in Fig. 12. We assume that Q contains only reachable states and λ contains only valid transitions.

Since we enriched the control-flow expression of the specification with decision variables in order to quantify the design space, the corresponding finite-state machine contains a representation of the design space according to the degrees of freedom introduced. Thus, we define now what we mean by an implementation of the finite-state machine M.

Definition 6.1: Let M be the finite-state machine obtained from a control-flow expression through derivation. We call M' an implementation of M if the following conditions hold.

1) The set of states of M' is a subset of the set of states of M.
2) The initial states of M and M' are the same.
3) The set of transitions of M' is a subset of the set of transitions of M.

Thus, an implementation $M' = (I, O, Q', \delta', \lambda', q_0)$ will be an implementation of $M = (I, O, Q, \delta, \lambda, q_0)$ if $Q' \subseteq Q, \delta' \subseteq \delta$ and $\lambda' \subseteq \lambda$. In addition to the requirements given above, we still require that M' also satisfies additional constraints that will be imposed by the structure of the original specification. We will present by an example the formulation of the multiprocess synthesis problem as an ILP. The complete formulation can be seen in Appendix B.

In the synthesis of M' from M, we have to identify which states will be included in M' and which transitions will be part of the transition function for M'. In order to determine the states of M which will be part of the states of M', we create a Boolean variable y_p for each state q_p of M. If the Boolean variable y_p is set to 1, our interpretation will be that the state q_p will belong to M'. We will denote the state q_p by p in the remainder of this section.

In order to determine a subset of the transitions of M', we subdivide each guard f of a transition $\delta(q_p, f)$ into two conjoined parts. The first part contains only decision variables and the second part contains only conditional variables. Let us call the first part $f_{\boldsymbol{X}}$ and the second part $f_{\boldsymbol{C}}$. Now, for each state q_p, decision variable x of $f_{\boldsymbol{X}}$ and for each different Boolean formula $f_{\boldsymbol{C}}$ of q_p, we create a Boolean variable $x_{(q_p, f_{\boldsymbol{C}})}$. In the solution of the ILP problem, the variables $x_{(q_p, f_{\boldsymbol{C}})}$ are assigned 0–1 values such that if $f_{\boldsymbol{X}}|_{x \leftarrow x_{(q_p, f_{\boldsymbol{C}})}} = 1$, then $\delta(q_p, f)$ belongs to M', i.e., if $f_{\boldsymbol{X}}$ evaluates to 1 when each variable x of $f_{\boldsymbol{X}}$ is assigned the value of $x_{(q_p, f_{\boldsymbol{C}})}$, then $\delta(q_p, f)$ will belong to M'.

Example 15: Let us consider the finite-state machine of Fig. 13 for the synchronization problem presented in Section II-1-1. For this finite-state machine, the set of mixed Boolean-ILP equations that quantifies the design space for the

decision variable x is shown below

$$\begin{aligned} y_0 &= 1 \\ y_1 - (x_{(0,1)}y_0 \vee x_{(2,c)}y_2) &= 0 \\ y_2 - y_1(\overline{x_{(1,c)}} \vee x_{(1,c)}) &= 0 \\ x_{(0,1)} &= 1 \\ x_{(1,\bar{c})} &= 1 \\ x_{(1,c)} + \overline{x_{(1,c)}} &= 1 \\ x_{(2,c)} &= 1 \\ (x_{(0,1)} \vee \overline{y_0})(x_{(1,c)}x_{(1,\bar{c})} \vee \overline{y_1})(x_{(2,c)} \vee \overline{y_2}) &= 0. \end{aligned}$$

The first set of equations represents the transition relation of M in terms of the decision variables and states. The first state of M (0) is always a state of M'. State 1 will be a state of M' if 0 is a state of M' and the transition $\delta(0,x)$ is in M', which is represented by assigning 1 to $x_{(0,1)}$; or if state 2 is a state of M' and the transition $\delta(2,xc)$ is in M', which is represented by assigning 1 to the Boolean variable $x_{(2,c)}$. A similar reasoning yields the third equation.

In the second group of equations, we represent the set of valid assignments for each state and conditional expression. The first equation states that the only possible choice for state 0 is to make a transition to state 1, and thus, $x_{(0,1)}$ should be assigned to 1. Similarly, when c is *false* on state 1, since the only possible choice is a transition to state 0, this transition should be a transition of M'. In the transition between states 1 and 2, there are two possible choices when c is *true*, and only one of those transitions should be assigned to M'.

In the third set of equations, we guarantee that for any causality constraint of the type $a \cdot (x : 0)^* \cdot b$, where a and b are actions and x is a decision variable, at least one state of M' will have x assigned to *false*, i.e., b will eventually be scheduled.

A assignment satisfying this set of equations is given by $y_0 = y_1 = y_2 = 1, x_{0,1} = x_{1,\bar{c}} = x_{2,c} = 1, x_{1,c} = 0$. □

6.1. Selection of a Cost Function

In the previous section, we considered just the formulation of the constraints to find an implementation of a finite-state representation. In system-level designs, we want to be able to distinguish possible implementations with respect to some cost measure in order to be able to select the optimal implementation. In addition, the designer should be able to add information about the environment. In our tool, the designer is allowed to control the synthesis solutions by specifying flexible objective functions, i.e., cost functions whose goals may be different for the different regions of the specification. For example, in a nested loop structure, the synthesis goal may be minimum delay for the inner loop, but minimum area for the outer loops. We will show here how to specify scheduling and binding cost functions by using actions and guards. Then, we will generalize the procedure by showing how the designer can specify more general objective functions with CFE's, whose goals change with the different regions of the specification.

6.1.1. Selecting Minimum Scheduling Costs: In synthesis, one of the primary goals is to obtain circuits whose running time is minimum. The selection of minimum scheduling costs using using control-flow expressions uses the following observation. *In single-source single-sink acyclic CDFG's, the synthesis of minimum schedules is equivalent to minimizing the execution time for the sink node of the CDFG.* Thus, every time an operation is delayed one cycle, we can associate an action 0 (corresponding to a delay of one cycle) that is inserted between the action which was delayed by one cycle and the action executed previously (Section IV). As a result, we can quantify the scheduling and causality constraints by counting the number of zeros inserted by the synthesis procedure.

The advantage of this method is that we may select fast schedules with respect to a restricted portion of the specification, or with respect to some set of conditionals, instead of just the minimum global schedule, giving more flexibility to the other parts of the specification.

We can express the scheduling cost of an implementation by considering the causality and scheduling constraints of the specification. For causality constraints of the type $(x : 0)^*$, where x is a decision variable, we cast the schedule cost as the number of times x is assigned to one, i.e., the amount of delay inserted due to decision variable x. Similarly, for a scheduling constraint of the form $(x^1 : 0 + x^2 : 0^2 + \cdots + x^n : 0^n)$, where $x^1, x^2, \cdots, x^n$ are decision variables. Every time x^i is assigned to one, the latency of the process in which x^i was specified is increased by i.

Example 16: In the Example 15, we can represent the scheduling cost on x by the cost $\mathbf{min}\ y_0x_{(0,1)} + y_1(x_{(1,c)}|x_{(1,\bar{c})}) + y_2x_{(2,c)}$, where $+$ denotes arithmetic addition and $|$ denotes Boolean disjunction.

This cost function represents all possible assignments x can have in the finite-state representation. Whenever x is assigned to one, corresponding to $x_{(1,c)}, x_{(0,1)}, x_{(1,\bar{c})}$, or $x_{(2,c)}$ being assigned to one, the execution time of process p_3 increases. Thus, any assignment to x that minimizes the number of times x is one over time (corresponding to the assignments of $x_{(1,c)}, x_{(0,1)}, x_{(1,\bar{c})}$, or $x_{(2,c)}$) reduces the latency of p_3.

The user specifies this cost function by requesting a minimization of the assignments of x over time, which can be automatically translated to the cost function given above. □

6.1.2. Selecting Minimum Binding Costs: In order to select a binding cost, we will have to define a partial cost function for actions, called here β. We then compute the disjunction of every transition of M' that contains action a, and weight this disjunction by $\beta(a)$.

Example 17: In Example 10, we rewrote the control-flow expression of the original specification in order to include binding constraints.

We can represent the binding cost of an implementation by the formula

$$\mathbf{min}\ \beta(M_1)[\vee_{T_{M_1}}] + \beta(M_2)[\vee_{T_{M_2}}] + \beta(M_3)[\vee_{T_{M_3}}]$$

where $\beta(M_i)$ is the cost of component M_i, and $\vee_{T_{M_i}}$ denotes the disjunction of all transitions of the implementation that contains M_i. Note that due to the complexity of the Boolean formula representing the disjunction of the set of transitions containing M_i, we decided not to put them explicitly here.

This formula states that the cost of M_i $(i \in \{1,2,3\})$ contributes to the cost of the implementation if at least one transition of M with output M_1 is a transition of M'. □

6.1.3. Generalizing Objective Functions: We showed previously how to select minimum scheduling and binding solutions by specifying their corresponding cost functions. We suggest, in this section, the combination of scheduling cost functions, binding cost functions, and control-flow expressions to obtain more general objective functions, such as the minimization of the execution time over paths or the minimization of the execution time of parts of a control-flow expression.

When we showed how scheduling and binding cost functions could be represented in our formulation, we only considered single transitions in the cost function. Because CFE's, decision variables and shadow actions can be used to represent constraints, we can combine constraint representation with objective functions and represent the cost of the whole path for an implementation. This combination provides the designer with the flexibility of further controlling the synthesis tool to change its goals according to the region being synthesized, or to guide the synthesis tool to introduce priorities in the synthesis process.

Example 18: In the specification of the ethernet coprocessor of Fig. 2, the transmission unit consists of three processes, *DMA_XMIT, XMIT_FRAME* and *XMIT_BIT*. Process *DMA_XMIT* receives a block as a byte stream from the bus and transmits it to the process *XMIT_FRAME*, which encapsulates the block with a frame and sends it to process *XMIT_BIT*. Thus, the transmission unit can represented by the control-flow expression $dma\_xmit||xmit\_frame||xmit\_bit$, with the appropriate synchronization corresponding to data transfers.

Let us consider the transmission of data from *dma_xmit* to *xmit_frame* to be represented by action a, the transmission of data from *xmit_frame* to *xmit_bit* to be represented by action b, and the initialization of the transmission command by action i. Thus, the expression $dma\_xmit||xmit\_frame||xmit\_bit||(x_0 : 0)^* \cdot i \cdot (x_1 : 0)^* \cdot a \cdot (x_2 : 0)^* \cdot b$ encapsulates with decision variables x_1 and x_2 all possible schedules of the transfers in the transmission unit. Thus, minimizing a cost function defined over the assignments to (x_1, x_2) will correspond to minimizing the execution time of the path that begins with the execution of the transmit data command, and ends at the transmission of the first bit. □

Note that the designer should be able to provide only cost measures by specifying which parts of the design he wants to tag with a cost function. The actual composition of the cost function and the computation of which transitions will be used in the cost function can be determined automatically.

6.2. Comparison with Other ILP Methods

We are going to analyze the procedure given above to obtain an implementation that minimizes or maximizes the cost functions defined above. We will compare the basic complexity of the algorithm with problems they are able to solve.

Most previous approaches to scheduling and binding are usually restricted to single-source, single-tail control-data flow graphs [4], [32], [36]–[40], [41], i.e., specifications in which the concurrent parts are restricted to begin at the same time, or to specifications which are dataflow intensive, as in the case of DSP's [42], [43]. Although those systems can also be synthesized with our approach, we further extend those methods by synthesizing the concurrent parts that may be running at different speeds or that may have complex interactions. In addition, we also consider the synchronization among the different parts of the system, which is only considered in a limited way in [3], [4]. In [44], a reconfiguration procedure for datapaths was described, but this reconfiguration is used only in case of failure.

Among the approaches to scheduling mentioned above, we are going to compare the execution time of our approach with exact methods using 0-1 integer linear programming formulations, such as [32], [38]. For single-source, single-tail control-data flow graphs, our method pays a penalty in the number of variables to be solved by the ILP solver, which is greater by a constant factor with respect to these other approaches. However, our methodology outperforms those other approaches in that it can handle loops, synchronization, and multirate execution of concurrent models.

If we consider the finite-state machine representation M of a control-flow expression p with n_s states, n_c conditionals and n_d decision variables, then the number of variables in the worst case will be on the order of $\mathcal{O}(n_s n_d 2^{n_c})$. Note, however, that in practical terms this upper bound is never reached, since not all decision variables will be evaluated in every state and not all possible expressions on conditionals are evaluated at the same time. If we compare this complexity with the complexity of the 0–1 ILP method of [32], we note that n_s related with the number of control-steps an operation can be scheduled in [32], n_d is related with the number of operations to be scheduled in [32] and $n_c = 1$ in [32], since no conditional paths can be specified in the formulation.

6.3. Solution of the ILP

In order to solve this set of 0–1 ILP equations, we developed a solver based on binary decision diagrams (BDD's) to obtain the set of solutions that minimizes/maximizes some cost function.

The reason for obtaining the set of solutions is that the user may be interested in further constraining a previous solution, or dynamically selecting which solution should be taken. For example, in a problem of determining which transaction should take a bus, the user may want to specify that during some cycles no transaction may be able to access the bus.

We refer the reader to [45] for an introduction on BDD's. BDD's have been used in several different applications, including the solution of 0–1 Integer Linear Programming [46], because of its low space complexity to represent some types of Boolean functions. In these problems, each equation of the 0–1 ILP problem is represented by a BDD. This BDD describes a function whose assignments to the Boolean variables satisfy the ILP equation. In a 0–1 ILP problem, the problem is specified as a set of equations and a cost function, that should be minimized or maximized. An assignment satisfying this set

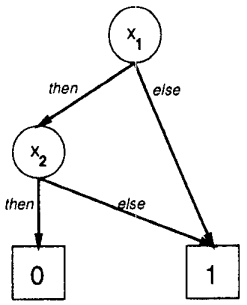

Fig. 15. BDD representing the constraint $4x_1 + 5x_2 \leq 8$.

of equations can be obtained by conjoining the BDD's for the different equations, and a solution that minimizes/maximizes a cost function can be obtained by a branch-and-bound on the set of valid assignments to the conjoined BDD with respect to the cost function.

Example 19: The equation $4x_1 + 5x_2 \leq 8$ is true by any assignment satisfying the Boolean formula $\overline{x}_1 \vee \overline{x}_2$, whose BDD is shown in Fig. 15. □

We developed a BDD-based ILP solver that extended the solution method presented in [46] by allowing equations not to be limited to linear equations on the Boolean variables, but to linear equations on Boolean functions over Boolean variables. Although the problems both solvers can solve are still the same, since the Boolean constraints can be represented by a set of linear separable equations, our solver has a smaller number of Boolean variables and equations to solve than the former when the equations include Boolean functions.

6.4. Derivation of Control-Unit

We now show how we can obtain an implementation for the original processes from the finite-state machine M'. Because M' was obtained by finding an implementation for the system $p = p_1||\cdots||p_n$ that minimizes some cost function, we can obtain a control-unit implementation for each p_i satisfying the assignments to the decision variables in M' by projecting these assignments into p_i. Thus, from the submachine $M' = (I, O, Q', \delta', \lambda', q_0)$, we construct machines $M_i = (I_i, O_i, Q', \delta_i, \lambda_i, q_0)$ for each concurrent part p_i of p.M_i will be the control-unit for this concurrent part of p.

The set I_i of inputs to M_i corresponds to the set of conditional variables of I. O_i corresponds to the multiset of actions of M_i. This multiset is a subset of O, restricted to the multisets of actions that can be generated from p_i alone. The transition function δ_i has the same transitions of δ', but with the set of inputs restricted to I_i. The output function λ_i is a restriction of λ' in such a way that the inputs are restricted to I_i and only the actions specified in p_i are maintained in λ_i.

Let us interpret this new transition function δ_i and the output function λ_i. Suppose we computed the finite-state machine representation N for p_i alone. In this finite-state machine representation, let us assume a state transition and an output generation that is dependent on some decision variable. After synthesizing the finite-state representation for p, and obtaining M_i, the transition of N was replaced by one or more transitions which depended only on the conditional variables. Even if the number of states in N and M_i does not agree, there will be equivalent transitions for N and M_i such that for each two equivalent states of N and M_i, there will be two corresponding transitions. Thus, this change in the transition function can be interpreted as if the decision variable of p_i were assigned the Boolean expression associated with the transition of δ_i. This mechanism can be used to dynamically reconfigure the system according to the system's state, based on the conditionals.

In practice, we would like to keep the number of possible schedules for a given operation small because dynamically scheduling an operation increases the complexity of the controller for a model. In our case, this was achieved by the following observations. First, a control-flow expression is unrolled only if it is necessary to generate a new state, since equivalent states are grouped together. Second, the controller obtained is a finite-state machine partially specified with respect to the conditionals whenever possible, because we leave room for sequential logic optimizers to further optimize the final controller.

Example 20: In the synchronization example discussed in previous examples, our goal is to obtain a control-unit implementation for p_3. Note that the assignment presented in Example 15 eliminates the transition from state 1 to state 2 when c is true. If we restrict the implementation on the actions generated by $p_3 = ((x : 0)^* \cdot a)^\omega$, we obtain the finite-state machine presented in Fig. 16. □

The reader should note that the finite-state machine we obtain by the procedure above does not guarantee any minimality with respect to the number of states, but just a finite-state machine that satisfies the original constraints and minimizes latency, which is the primary optimization goal. We use the state minimizer *Stamina* [47] to obtain the minimum number of states for the control-flow expression. In fact, in Example 20, an implementation with minimum number of states can be obtained with just two states.

Note that the complexity of the finite-state machine for each of the control-flow expressions will have a complexity of the product machine in the worst case, i.e., when the amount of synchronization among the machines is high. However, if the amount of synchronization among several control-flow expressions is high, then the number of states of the finite-state machine will be much lower than the product of the number of states for the finite-state machine of the individual control-flow expressions. Thus, we do not expect the final complexity of the machines to be much higher, except for the subparts that are not tightly coupled. Note, however, that we can always find transformations of the control-flow expression that maximize the tightly coupled regions of a control-flow expression [48].

VII. Implementation and Results

We implemented a program to synthesize controllers with dynamic schedules from control-flow expressions in 20 000 lines of C, and a 0–1 ILP solver using BDD's in 3000 lines of C.

Since the technique presented in this paper is targeted for the synthesis of concurrent systems under synchronization, which

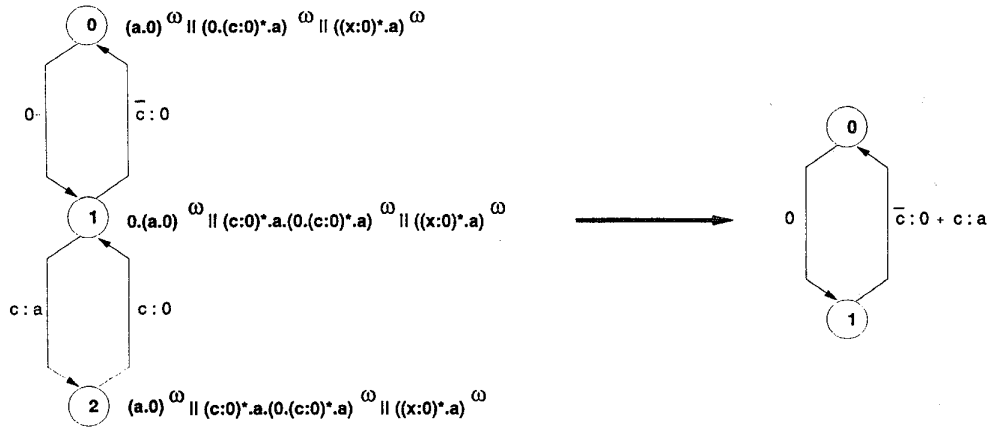

Fig. 16. Finite-state machine for control-flow expression $((x : 0)^* \cdot a)^\omega$.

is a new area, there are no standard benchmarks yet. Thus, instead of comparing our approach with the existing techniques for scheduling and binding using standard benchmarks, we will show an application of this technique for designing the circuits described in Section II.

7.1. Applying Scheduling Constraints to the Ethernet Coprocessor

We consider here the Ethernet coprocessor of Fig. 2. In that figure, let us focus on the transmission unit. As mentioned in Example 18, the transmission unit is composed by three processes, *dma_xmit, xmit_frame* and *xmit_bit*. Upon receiving a byte from process *xmit_frame, xmit_bit* sends the corresponding bit streams over the line TXD. Thus, *xmit_bit* must receive each byte eight cycles apart, which constraints the rate in which the bytes are transmitted from *xmit_frame*.

Process *xmit_frame* was specified as a program state machine written in Verilog HDL, as shown in Fig. 17, and it was also specified with an exception handling mechanism, i.e., the *disable* command of Verilog HDL. We refer the reader to [48] for additional details on the implementation. Table II presents the results for the scheduling of *xmit_frame* from its control-flow expression model. The first column shows the number of states of *xmit-frame* before scheduling the operations. The second column shows the number of states after state minimization. The third column shows the size of the constraints in terms of BDD nodes, used by the BDD ILP solver. The fourth column shows the execution time taken to obtain a satisfying schedule minimizing the execution time of the process. Note that by having a finite-state representation of the behavior of the system in two different implementations, we were able to obtain two comparable implementations in the number of states.

7.2. Protocol Conversion for a PCI Bus

We implemented the four models for the reading and writing cycles of the PCI local bus and the SDRAM mentioned in Section II-1-2 in 230 lines of a high-level subset of Verilog HDL, with the corresponding CFE's having similar complexity. These models are predefined libraries that can synchronize with any circuit. We thus use the technique of synchronization

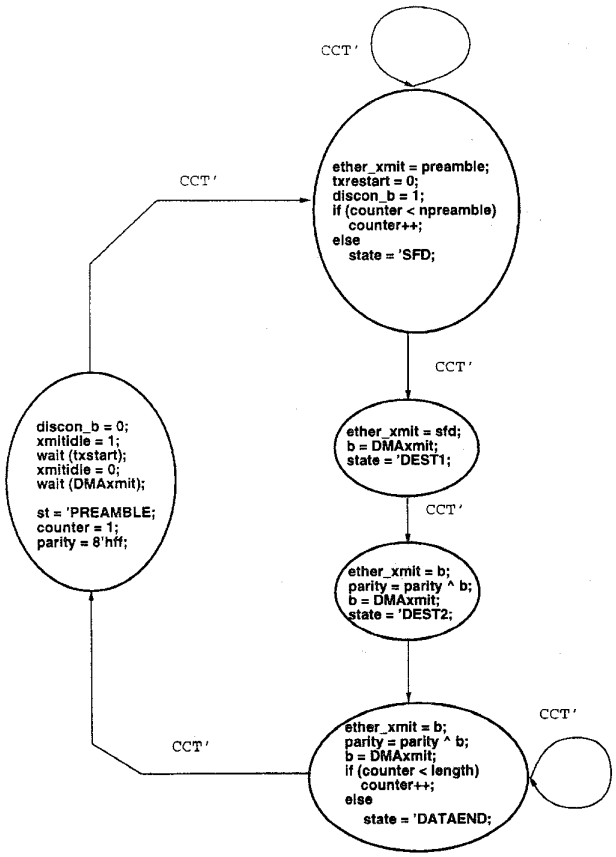

Fig. 17. Program state machine for process *xmit_frame*.

TABLE II
RESULTS OF THE $xmit\_frame$ SYNTHESIS PROBLEM

	# States		Constraint	Time
xmit-frame (except.)	178	90	324	15.7s
xmit-frame	178	90	989	87s

TABLE III
PCI/SDRAM PROTOCOL CONVERSION EXAMPLE

Model	States PCI	States SDRAM	States PCI + SDRAM	Execution Time	Actions	Conditionals	Decision Variables
READ	7	15	34	3.5 s	16	8	6
WRITE	6	7	30	1.6 s	15	8	3

synthesis in order to synthesize a combined controller that is smaller than the two separate controllers.

Table III shows the number of states for the controllers in terms of a Mealy machine, when each part is synthesized separately, and when the controller for both models is generated as a single controller, which is highly desirable, since both parts are highly synchronized. Although the number of states in the single controller is higher than the number of states used when both specifications are synthesized separately, the total number of registers used is smaller, due to the reduction of unreachable states of both specifications. (For example, a SDRAM transfer does not occur if the PCI is not also transferring data.) We also show the number of actions, conditionals, and decision variables for both descriptions. In both cases, we attempted to minimize the execution time of the combined description.

VIII. SUMMARY AND CONCLUSION

We considered in this paper the modeling and synthesis problems for specifications that are better described as a set of concurrent and interacting parts, or *multiprocess* descriptions. For these specifications, current synthesis tools achieve suboptimal results, due to the local scope of such tools, i.e., they do not consider the reconfigurability of one part of a specification with respect to the other parts.

In order to capture the degrees of freedom available in such designs most effectively, we developed a modeling technique for control-flow dominated specifications, and we presented a methodology for automatically obtaining the controllers for the concurrent parts of the specification.

Modeling was performed in the algebraic domain, which we called the algebra of control-flow expressions. Using control-flow expressions, the system was abstracted in terms of its control-flow. Control-flow expressions were manipulated algebraically with operations such as term rewriting, and synchronization specification. Constraints were also represented as control-flow expressions, which allowed an uniform method for representing the specification and constraints.

Synthesis was performed in the state-space domain. We showed how a control-flow expression can be translated into a finite-state representation, where the analysis and synthesis tasks were performed. The conversion from a control-flow into a finite-state machine was achieved by computing the derivatives of a control-flow expression. We showed that the number of derivatives was finite, and that only a finite number of iterations was necessary to obtain all the derivatives of an expression.

Analysis of a control-flow expression was performed by checking for emptiness of the corresponding finite-state machine representation. This allowed us to check if a specification was overconstrained, and thus conclude that no solution existed for the synthesis problem.

Synthesis was cast as an 0–1 ILP problem. In the ILP formulation, the designer was allowed to specify flexible objective functions in order to have a better control over the synthesis procedure. These functions allowed the different regions of computation of a system-level design to have different goals, which were satisfied during the synthesis of the specification.

The ILP problem was solved using a BDD solver. Among the advantages of this solver was that it included the reduced number of variables that needed to be handled and the capability of considering intermediate solutions, i.e., the capability of adding synchronization elements at the end of the synthesis process to allow for extensibility of the design.

Reconfigurability of a process with respect to the process' environment was achieved by allowing an assignment to a decision variable to vary over time. Thus, at different states of the system we were able to obtain different assignments to the decision variables.

As future work, we are currently investigating possible extensions to control-flow expressions. Among them, we are considering the specification of constraints as negations of CFE's, addition of an exception handling mechanisms to CFE's, and the incorporation of internal variables. Since the size of the finite-state machine for each control-flow expression depends heavily on the amount of synchronization, we intend to use this fact to reduce the size of a finite-state machine when synthesizing the finite-state machine for a control-flow expression, and to facilitate the specification of constraints. With this new method, the complexity of the intermediate representation would be further reduced. In addition, the synchronization of the different parts can be further reduced by considering synchronization only at small blocks or subparts of the specification, instead of considering the full specification. Finally, we are currently investigating the use the algebra of control-flow expressions to perform high-level restructuring of the control-flow.

APPENDIX A
DERIVATIVES

We show in this appendix how we can use the computation of derivatives to compute the suffixes of a CFE, and that derivatives of a CFE correspond to the cycle-by-cycle simulation of the CFE. In order to define derivatives of a control-flow expression, we need to know if the control-flow expression can execute in zero time. Thus, we define a function Δ that returns a Boolean expression over the set of conditionals and decision variables for those guards that enable zero-cycle paths (or ϵ-paths) in a CFE.

Definition A.1: Let $\Delta : \mathcal{F} \rightarrow \mathcal{G}$ be a function defined recursively as follows:

1)

$$\Delta(f:\epsilon) = f, where\ f \in \mathcal{G}$$
$$\Delta(\delta) = 0$$
$$\Delta(a) = 0, where\ a \in \mathcal{M}^{\mathcal{A}}.$$

2) Let $P, Q \in \mathcal{F}$ and let $\Delta(P)$ and $\Delta(Q)$ be the guards that generate ϵ in P and Q, respectively. We assume that if $c_1, c_2, g \in \mathcal{G}$, and that for any two guards f and g, fg is the conjunction of f and g, that $f|g$ is the disjunction of f and g and that $\overline{f}$ is the negation of f

$$\Delta(P \cdot Q) = \Delta(P)\Delta(Q)$$
$$\Delta(c_1 : P + c_2 : Q) = c_1\Delta(P)|c_2\Delta(Q)$$
$$\Delta((g:P)^*) = \overline{g}$$
$$\Delta(P^\omega) = 0$$
$$\Delta(P||Q) = \Delta(P)\Delta(Q).$$

The function Δ determines which assignment to conditionals and decision variables make a control-flow expression execute ϵ, that executes in zero time. Assume for some CFE $p, \Delta(p) \neq 0$. If we compose p inside a loop $((c : p)^*)$ or in an infinite computation $(p^\omega), (c : p)^*$ and p^ω will violate the synchrony hypothesis and the synchronous execution semantics we defined earlier, since in $(c : p)^*$, or similarly for p^ω, there is at least one assignment to the guards that would make c be evaluated consecutively in the same clock cycle.

Definition A.2: Let p be a control-flow expression. We say $(c : p)^*$ and p^ω are well-formed CFE's (WFCFE's) if $\Delta(p) = 0$.

Although nonWFCFE's appear in real life specifications, synthesis tools always make the assumption that each loop or infinite repetition will take at least one cycle. Thus, we must be able to convert nonWFCFE's into WFCFE's such that the execution time for the non-ϵ executions is maintained, and a delay is generated for ϵ executions.

Theorem A.1: Let $\Delta(p) \neq 0$, for some CFE p. Then $\Delta(p \cdot (\Delta(p) : 0 + \overline{\Delta(p)} : \epsilon)) = 0$.

Proof: $\Delta(p \cdot (\Delta(p) : 0 + \overline{\Delta(p)} : \epsilon)) = \Delta(p)\overline{\Delta(p)} = 0$ ■

Note that for any other assignment to the conditionals and decision variables such that the ϵ is not executed in p, an ϵ is executed in $(\Delta(p) : 0 + \overline{\Delta(p)} : \epsilon)$. We have thus shown that for any nonWFCFE, we can obtain an equivalent CFE which is well-formed. Thus, we will consider in this appendix CFE's which are WFCFE's, since they will correspond to circuits that will be implemented.

The derivatives of a CFE correspond to a cycle-by-cycle simulation of the CFE. Since actions in a control-flow expression have a single-cycle semantics, a cycle-by-cycle simulation of a control-flow expression is equivalent to extracting all actions that can be executed next from a control-flow expression.

We will represent the derivative of a control-flow expression by the operator ∂. This operator, when applied to a CFE, yields a triple in $\mathcal{G} \times \mathcal{M}^{\mathcal{A}} \times \mathcal{F}$, where $\mathcal{G}$ is the set of Boolean expressions over the set of conditional and decision variables, $\mathcal{M}^{\mathcal{A}}$ is the set consisting of multisets of actions, and $\mathcal{F}$ is the set of control-flow expressions. The triple $(\gamma, \mu, \pi) \in \mathcal{G} \times \mathcal{M}^{\mathcal{A}} \times \mathcal{F}$ obtained from a CFE p indicates that the actions μ are executed when γ is true, followed by the execution of π.

Definition A.3: Let $\partial : \mathcal{F} \rightarrow (\mathcal{G} \times \mathcal{M}^{\mathcal{A}} \times \mathcal{F})$ be defined as a derivative of a control-flow expression, given recursively as follows:

$$\partial(f:\epsilon) = \{(f, \epsilon, \epsilon)\}$$
$$\partial(f:a) = \{(f, a, \epsilon)\}$$
$$\partial(\delta) = \emptyset, \text{ the empty set}$$
$$\partial(f:0) = (f, 0, \epsilon)$$
$$\partial(p \cdot q) = \{(\gamma, \mu, \pi \cdot q) | (\gamma, \mu, \pi) \in \partial p\} \cup \{(\Delta(p)\gamma, \mu, \pi) | (\gamma, \mu, \pi) \in \partial q\}$$
$$\partial(f_1 : p + f_2 : q) = \{(f_1\gamma, \mu, \pi) | (\gamma, \mu, \pi) \in \partial p \wedge (\mu \neq \epsilon)\} \cup \{(f_2\gamma, \mu, \pi) | (\gamma, \mu, \pi) \in \partial q \wedge (\mu \neq \epsilon)\}$$
$$\partial(p^\omega) = \{(\gamma, \mu, \pi) | (\gamma, \mu, \pi) \in \partial(p \cdot p^\omega)\}$$
$$\partial((f:p)^*) = \{(f\gamma, \mu, \pi) | (\gamma, \mu, \pi) \in \partial(p \cdot (f:p)^*)\}$$
$$\partial(p||q) = \{(\gamma_p \wedge \gamma_q, \mu_p \cup \mu_q, \pi_p||\pi_q) | (\gamma_p, \mu_p, \pi_p) \in \partial p \wedge (\gamma_q, \mu_q, \pi_q) \in \partial q\}.$$

Example 21: Let $p = (a \cdot b \cdot c)^\omega$.

$$\partial p = \{(\gamma, \mu, \pi) | (\gamma, \mu, \pi) \in \partial(a \cdot b \cdot c \cdot (a \cdot b \cdot c)^\omega\}$$
$$= \{(\gamma, \mu, \pi \cdot b \cdot c \cdot (a \cdot b \cdot c)^\omega) | (\gamma, \mu, \pi) \in \partial a\} \cup \emptyset$$
$$= \{(true, a, b \cdot c \cdot (a \cdot b \cdot c)^\omega)\}.$$

Thus, after the first cycle in which action a is executed, p transforms into $b \cdot c \cdot (a \cdot b \cdot c)^\omega$. □

Now, let us extend the definition of ∂ operator to the iterative application of ∂ to a control-flow expression. Since we can consider each application of ∂ as a one-cycle simulation of the control-flow expression, then the iterative application of ∂ corresponds to a multicycle simulation of the control-flow expression.

Definition A.4: Let p be a control-flow expression. $\partial^i p$ is defined recursively as follows:

$$\partial^1 p = \partial p$$
$$\partial^i p = \bigcup_{(\gamma,\mu,\pi) \in \cup_{j=1}^{i-1} \partial^j p} \partial\pi.$$

Let us now define formally what is a suffix of a control-flow expression.

Definition A.5: Let p be a control-flow expression. Then q is a suffix of p if $q = p$ or if $\exists n, \gamma, \mu : (\gamma, \mu, q) \in \partial^n p$.

The definition above allows the following formula to be used for computing the set of suffixes of a control-flow expression

$$\text{Suffixes}\,(p) = \cup_{n=1}^{\infty}\{\pi | (\gamma, \mu, \pi) \in \partial^n p\} \cup \{p\}.$$

Although the formula presented above computes all the suffixes of a control-flow expression, the formula does not specify that the number of suffixes is finite, and neither does it specify that the set of suffixes can be obtained after a

finite number of iterations. Thus, we have to show that this procedure is in fact effective, i.e., that it will terminate after a finite number of iterations.

In order to show that the number of suffixes is finite, we first have to eliminate any two suffixes that are equivalent, according to the following definition.

Definition A.6: Two control-flow expressions p and q are equivalent if one can be obtained from the other using the CFE axioms (Table I).

Example 22: The control-flow expression $(a \cdot b \cdot c)^\omega$ is equivalent to $a \cdot b \cdot c \cdot (a \cdot b \cdot c)^\omega$.

Thus we will only consider the set of suffixes for a control-flow expression such that no two suffixes are equivalent. This set of suffixes will be called the set of *irredundant suffixes* of a control-flow expression. In the rest of this paper, we will refer to the set of irredundant suffixes of a control-flow expression just by the set of suffixes of the control-flow expression.

The following theorem shows that the number of derivatives of a control-flow expression is finite, considering that any two equivalent control-flow expressions are represented by the same set element during the computation of a derivative.

Theorem A.2: Every control-flow expression p has a finite number of derivatives, i.e., $|\cup_{i=0}^{\infty} \partial^i p|$ (the number of elements of this set) is finite.

Proof: We are going to prove this theorem recursively on the number of CFE compositions.

1) **Basis:** $|\cup_{i=0}^{\infty} \partial^i (f : a)| \leq 2$ and $|\cup_{i=0}^{\infty} \partial^i \delta| = 0$
2) **Inductive Step:** Let $|\cup_{i=0}^{\infty} \partial^i p| \leq N_p$ and $|\cup_{i=0}^{\infty} \partial^i q| \leq N_q$ for control-flow expressions p and q

$$
\begin{aligned}
\left|\cup_{i=0}^{\infty}\partial^i(p \cdot q)\right| &\leq \left(\left|\cup_{i=0}^{\infty}\partial^i p\right| \times \left|\cup_{i=0}^{\infty}\partial^i q\right|\right) \\
&\quad + \left|\cup_{i=0}^{\infty}\partial^i(\Delta(p) : q)\right| \\
&\leq N_p N_q + N_q \\
\left|\cup_{i=0}^{\infty}\partial^i(c_1 : p + c_2 : q)\right| &\leq \left|\cup_{i=0}^{\infty}\partial^i(c_1 : p)\right| \\
&\quad + \left|\cup_{i=0}^{\infty}\partial^i(c_2 : q)\right| \\
&\leq N_p + N_q \\
\left|\cup_{i=0}^{\infty}\partial^i(p||q)\right| &\leq \left|\cup_{i=0}^{\infty}\partial^i p\right| \times \left|\cup_{i=0}^{\infty}\partial^i q\right| \\
&\leq N_p N_q \\
\left|\cup_{i=0}^{\infty}\partial^i(p^\omega)\right| &\leq \left|\cup_{i=0}^{\infty}\partial^i(p \cdot p^\omega)\right| \\
&\leq 2\left|\cup_{i=0}^{\infty}\partial^i p\right| \\
&\leq 2N_p \\
\left|\cup_{i=0}^{\infty}\partial^i((c : p)^*)\right| &\leq \left|\cup_{i=0}^{\infty}\partial^i(c : p.(c : p)^*)\right| \\
&\leq 2\left|\cup_{i=0}^{\infty}\partial^i(p.(c : p)^*)\right| \\
&\leq 2N_p. \qquad \blacksquare
\end{aligned}
$$

Theorem A.3: For any control-flow expression p, there exists N such that for all $M > N$ $\cup_{i=0}^{N} \partial^i p = \cup_{i=0}^{M} \partial^i p$.

Proof: Suppose $\cup_{i=1}^{N} \partial^i p = \cup_{i=1}^{N-1} \partial^i p$, but $\cup_{i=1}^{N+1} \partial^i p \neq \cup_{i=1}^{N} \partial^i$, where N is the least integer in which this occurs.

Since $\cup_{i=0}^{N+1} \partial^i p = (\cup_{i=0}^{N} \partial^i p) \cup (\partial^{N+1} p)$. Then $\partial^{N+1} p$ must contain some derivative not included in $\cup_{i=0}^{N}$.

We defined $\partial^{N+1} p = \cup_{(\gamma,\mu,\pi)\in\cup_{i=1}^{N} \partial^i p} \pi$. Note that we have $\cup_{i=1}^{N} \partial^i p = \cup_{i=1}^{N-1} \partial^i p$. Thus, $\partial^{N+1} p = \cup_{(\gamma,\mu,\pi)\in\cup_{i=1}^{N-1} \partial^i p} \pi = \partial^N p$, and $\cup_{i=0}^{N+1} \partial^i p = \cup_{i=0}^{N} \partial^i p$, a contradiction.

In summary, we presented a way to compute all the suffixes of a control-flow expression. We also showed that the number of suffixes is finite, since the number of derivatives is finite, and that only a finite number of derivatives is necessary to obtain the sets of suffixes.

Appendix B
Formulation of Multisynthesis Problem as 0–1 ILP Instance

We formulate the problem of finding an implementation for the finite-state representation as an ILP instance. We will use here $x, f_{\boldsymbol{X}}, c$ and $f_{\boldsymbol{C}}$, as defined in Section VI. Let us define also $f_{\boldsymbol{X}}^{\boldsymbol{C}}$ which stands for $(\forall x \in \boldsymbol{x}) f_{\boldsymbol{X}}(x = x_{p,f_{\boldsymbol{C}}})|_x$, i.e., the formula obtained by replacing every occurrence of $x \in f_{\boldsymbol{X}}$ by $x_{p,f_{\boldsymbol{C}}}$.

Finally, let $X = \{x_{p,f_{\boldsymbol{C}}}\} \cup \{y_p\}$ be the set of all Boolean variables defined previously for the finite-state machine M. We want to obtain an assignment to the variables in X such that the following set of equations hold.

- The initial state of the finite-state machine M is a valid state of every implementation M' of M: $y_p = 1$, where p denotes here the original control-flow expression.
- Each state p' of M is a state of M' ($y'_p = 1$) if for every transition to p' $((p, f_{\boldsymbol{X}} f_{\boldsymbol{C}}, p') \in P), y'_p = \vee_p y_p f_{\boldsymbol{X}}^{\boldsymbol{C}}$, i.e., p' is a state of M' if there is a state p that is a state of P', and there is an assignment to X such that $f_{\boldsymbol{X}}^{\boldsymbol{C}} = 1$.
- For each alternative composition in which the guards are decision variables—such as in the case of the module selection or in scheduling constraint specification—only one decision variable should be *true* for a given state of the finite-state machine.

  This statement is captured by following formula: $\Sigma x_{p,f_{\boldsymbol{C}}} = 1$, for all p and transitions $\delta(p, f) = p'$ and $\lambda(p, f) = a$ such that $x \in \boldsymbol{x}$, and Σ denotes the arithmetic addition.
- For each causality constraint $(x : 0)^*$, where x is a decision variable, we assume that eventually the computation should proceed. In other words, there is at least one state of the implementation M' in which x should be different from one.

The following equation captures this constraint: $\wedge_{\delta(p,f)=p' \wedge \lambda(p,f)=a} (\wedge x_{p,f_{\boldsymbol{C}}} \vee y_p) = 0$.

Acknowledgment

The authors wish to thank the anonymous reviewers for their many helpful comments.

References

[1] G. De Micheli, *Synthesis and Optimization of Digital Circuits*. New York: McGraw Hill, 1994.
[2] G. Berry and G. Gonthier, *The Esterel Synchronous Programming Language: Design, Semantics, Implementation*. Paris, France: Ecole Nationale Supérieure des Mines de Paris and Institut National de Recherche en Informatique et Automatique, 1988.
[3] Y.-H. Hung and A. C. Parker, "High-level synthesis with pin constraints for multiple-chip designs," in *Proc. Design Automation Conf.*, June 1992, pp. 231–234.

[4] D. Filo, D. C. Ku, C. N. Coelho Jr., and G. De Micheli, "Interface optimization for concurrent systems under timing constraints," *IEEE Trans. VLSI Syst.*, vol. 1, pp. 268–281, Sept. 1993.
[5] *PCI Local Bus Specification Revision 2.1*, 1995.
[6] G. Borriello and R. Katz, "Synthesis and optimization of interface transducer logic," in *Proc. Int. Conf. Computer-Aided Design*, Santa Clara, CA, Nov. 1987, pp. 56–60.
[7] G. Borriello, "A new interface specification methodology and its application to transducer synthesis," Univ. California, Berkeley, CA, UCB/CSD Tech. Rep. (dissertation) 88/4301988, 1988.
[8] D. Ku and G. De Micheli, *High-level Synthesis of ASIC's Under Timing and Synchronization Constraints*. Norwell, MA: Kluwer Academic, 1992.
[9] S. Narayan and D. Gajski, "Interfacing incompatible protocols using interface process generation," in *Proc. Design Automation Conf.*, June 1995, pp. 468–473.
[10] J. Nestor and D. Thomas, "Behavioral synthesis with interfaces," in *Proc. Design Automation Conf.*, June 1986, pp. 112–115.
[11] R. K. Gupta, C. N. Coelho Jr., and G. De Micheli, "Synthesis and simulation of digital systems containing interacting hardware and software components," in *Proc. 29th Design Automation Conf.*, June 1992, pp 225–230.
[12] ——, "Program implementation schemes for hardware-software systems," *IEEE Trans. Comput.*, vol. 43, pp. 48–55, Jan. 1994.
[13] A. Wu, D. Gajski, N. Dutt, and S. Lin, *High-Level VLSI Synthesis—Introduction to Chip and System Design*. Norwell, MA: Kluwer Academic, 1992.
[14] Z. Zhu and S. D. Johnson, "An example of interactive hardware transformation," Indiana Univ., Tech. Rep. 383, 1993.
[15] D. E. Thomas and P. R. Moorby, *The Verilog Hardware Description Language*. Norwell, MA: Kluwer Academic, 1991.
[16] F. Boussinot and R. De Simone, "The ESTEREL language," *Proc. IEEE*, vol. 79, pp. 1293–1303, Sept. 1991.
[17] J. C. M. Baeten, *Process Algebra*. New York: Cambridge Univ. Press, 1990.
[18] J. E. Hopcroft and J. D. Ullman, *Introduction to Automata Theory, Languages and Computation*. Reading, MA: Addison Wesley, 1979.
[19] J. Laski, "Path expressions in data flow program testing." in *Proc. 14th Ann. Int. Comput. Software Appl. Conf.*, Chicago, IL, 1990, pp. 570–576.
[20] M. R. Paige, "On partitioning program graphs," *IEEE Trans. Software Eng.*, vol. SE-3, pp. 386–393, Nov. 1977.
[21] R. Milner, "A complete inference system for a class of regular behaviors," *J. Comput. Syst. Sci.*, vol. 28, pp. 439–467, 1984.
[22] Y. Choueka, "Theories of automata on ω-tapes: A simplified approach," *J. Comput. Syst. Sci.*, vol. 8, pp. 117–141, 1974.
[23] R. H. Campbell, "Path expressions: A technique for specifying process synchronization," Ph.D. dissertation, Dept. Comput. Sci., Univ. Illinois, Urbana, IL, UIUCDCS-R-77-863, Aug. 1976.
[24] D. Drusinsky and D. Harel, "Statecharts as an abstract model for digital control-units," Weizmann Inst. Sci., Tech. Rep. CS86-12, 1986.
[25] R. Saracco, *Telecommunications Systems Engineering Using SDL*. New York: Elsevier Sci., 1989.
[26] J. L. Peterson, *Petri Net Theory and the Modeling of Systems*. Englewood Cliffs, NJ: Prentice-Hall, 1981.
[27] T. H. Wang, "Repeatable firing sequences for petri nets under conventional, subset and timed firing rules," Ph.D. dissertation, Case Western Reserve Univ., Cleveland, OH, 1988.
[28] R. Milner, "Handbook of theoretical computer science," *Operational and Albebraic Semantics of Concurrent Processes* Cambridge, MA: Mass. Inst. Technol. Press, 1991, vol. 2, ch. 19, pp. 1201–1242.
[29] Z. Zhu and S. D. Johnson, "Automatic synthesis of sequential synchronization," in *Proc, IFIP Conf. Hardware Descript. Lang. Appl.*, Ottawa, Canada, Apr. 1993.
[30] ——, "Capturing synchronization specifications for sequential compositions," in *Proc. Int. Conf. Comput. Design*, Oct. 1994, pp. 117–121.
[31] W. Wolf, A. Takach, and T. Lee, "High-Level VLSI Synthesis," *Architectural Optimization Methods for Control-Dominated Machines*. Norwell, MA: Kluwer Academic, 1991.
[32] C.-T. Hwang, J.-H. Lee, and Y-C Hsu, "A formal approach to the scheduling problem in high-level synthesis, *IEEE Trans. Computer-Aided Design*, vol. 10, pp. 464–475, Apr. 1991.
[33] A. Takach, W. Wolf, and M. Leeser, "An automaton model for scheduling constraints, *IEEE Trans. Comput.*, vol. 44, pp. 1–12, Jan. 1995.
[34] W. Wolf, A. Takach, C. Huang, and R. Manno, "The Princeton University behavioral synthesis system," in *Proc. 29th Design Automation Conf.*, June 1992, pp. 182–187.
[35] J. A. Brzozowski, "Derivatives of regular expressions," *J. Assoc. Comput. Mach.*, vol. 11, no. 4, pp. 481–494, Oct. 1964.
[36] C. H. Gebotys, *Optimal VLSI Architectural Synthesis*. Norwell, MA: Kluwer Academic, 1991.
[37] T. Kim, J. W. S. Liu, and C. L. Liu, "A scheduling algorithm for conditional resource sharing," in *Proc. Int. Conf. Computer-Aided Design*, Santa Clara, CA, Nov. 1991, pp. 84–87.
[38] I. Radivojević and F. Brewer, "Symbolic techniques for optimal scheduling," in *Proc. Synth. Simul. Meeting Int. Interchange—SASIMI*, Nara, Japan, Oct. 1993, pp. 145–154.
[39] L. Hafer and A. Parker, "Automated synthesis of digital hardware," *IEEE Trans. Comput.*, C-31, pp. 93–109, Feb. 1982.
[40] P. Marwedel, "Matching system and component behavior in mimola synthesis tool," in *Proc. Euro. Design Automation Conf.*, Mar. 1990, pp. 146–156.
[41] K. Wakabayashi and H. Tanaka, "Global scheduling independent of control dependencies based on condition vectors," in *Proc. Design Automation Conf.*, June 1992, pp. 112–115.
[42] D. Lanneer *et al.*, "Architectural synthesis for medium and high throughput signal processing with the new cathedral environment," in R. Camposano and W. Wolf, Eds., *High-Level {VLSI} Synthesis*. Norwell, MA: Kluwer Academic, June 1991, pp. 27–54.
[43] S. Note, G. Goossens, F. Catthoor, and H. De Man, "Combined hardware selection and pipelining in high-performance data-path design," *IEEE Trans. Computer-Aided Design*, vol. 11, pp. 413–423, Apr. 1992.
[44] L. Guerra, M. Potkonjak, and J. Rabaey, "High level synthesis for reconfigurable datapath structures," in *Proc. Int. Conf. Computer-Aided Design*, Nov. 1993, pp. 26–29.
[45] R. E. Bryant, "Graph-based algorithms for Boolean function manipulation," *IEEE Trans. Comput.*, vol. C-35, pp. 677–691, Aug. 1986.
[46] S.-W. Jeong and F. Somenzi, "Logic synthesis and optimization," *A New Algorithm for 0–1 Programming Based on Binary Decision Diagrams*. Norwell, MA: Kluwer Academic, 1993, ch. 2, pp. 145–166.
[47] J.-K. Rho, G. D. Hachtel, F. Somenzi, and R. M. Jacoby, "Exact and heuristic algorithms for the minimization of incompletely specified state machines," *IEEE Trans. Computer-Aided Design*, vol. 13, pp. 167–177, Feb. 1994.
[48] C. N. Coelho Jr., "Analysis and synthesis of concurrent digital circuits using control-flow expressions," Ph.D. dissertation, Stanford Univ., Stanford, CA, 1995.

Claudionor Nunes Coelho, Jr. received the B.S. degree in electrical engineering (summa cum laude) and the M.Sc. degree in computer science from the Federal University of Minas Gerais, (UFMG), Brazil, in 1988 and 1990, respectively. He also received the Ph.D. degree in electrical engineering/computer science from Stanford University, Stanford, CA, in 1996.

He is an Adjunct Professor with the Department of Computer Science at the UFMG. His research interests include the specification, synthesis, and formal verification of hardware and software components for real-time embedded systems, computer architecture, and performance evaluation of hardware-software systems. From 1993 to 1996, he was a summer intern and a staff member on formal hardware verification with Fujitsu Laboratories of America and with Integrated Information Technology.

Giovanni De Micheli (S'79–M'82–SM'89–F'89) for a photograph and biography, see p. 643 of the June 1996 issue of this TRANSACTIONS.