1

MODELING AND SYNTHESIS OF SYNCHRONOUS SYSTEM-LEVEL SPECIFICATIONS

Claudionor Nunes Coelho Jr*, Giovanni De Micheli**

* *Department of Computer Science, Computer Engineering Laboratory, Universidade Federal de Minas Gerais, Belo Horizonte, MG, Brazil.*

** *Computer Systems Laboratory, Stanford University, Stanford, CA 94305.*

ABSTRACT

We present in this chapter a modeling style and control synthesis technique for system-level specifications that are better described as a set of concurrent descriptions, their synchronizations and complex constraints. For these types of specifications, conventional synthesis tools will not be able to enforce design constraints because these tools are targeted to sequential components with simple design constraints.

In order to schedule operations satisfying the constraints of system-level specifications, we propose a synthesis tool called Thalia that considers the degrees of freedom introduced by the concurrent models and by the system's environment.

The synthesis procedure is subdivided into the following steps: we first model the specification in an algebraic formalism called control-flow expressions, that considers most of the language constructs used to model systems reacting to their environment, i.e. sequential, alternative, concurrent, iterative, and exception handling behaviors. Such constructs are found in languages such as C, Verilog HDL, VHDL, Esterel and StateCharts.

Then, we convert this model and a suitable representation for the environment into a finite-state machine, where the system is analyzed, and design constraints such as timing, resource and synchronization are incorporated. The operations in this representation are scheduled using a 0-1 Integer Linear Programming solver implemented with Binary Decision Diagrams.

1.1. INTRODUCTION

The use of synthesis tools in synchronous digital designs at the logic and higher levels has gained large acceptance in industry and academia. Three of the reasons for its acceptance are the increasing complexity of the circuits, the need for reducing time to market and the requirement to design circuits correctly and optimally. In order to meet these requirements of today's marketplace, designers have to rely on the ability to specify their designs at higher levels of abstraction. In particular, designers depend upon models that describe the specification at a level higher than logic level and RTL level [1].

Above the logic level of abstraction, circuit designs have been described at high-level and system-level. We denote by high-level abstraction the modeling style based on the representation of a circuit design by blocks of operations and their dependencies. High-level abstraction has been used effectively for representing designs in digital signal processing applications [2]. However, when representing designs that are better specified as a set of concurrent and interacting components, this abstraction level will not be able to capture the synchronization introduced by the components executing concurrently.

We call system-level abstraction a modeling style based on the description of concurrent and interacting modules and system-level synthesis the corresponding task of deriving a logic-level description from such a model. Concurrency allows designers to reduce the complexity by partitioning the circuit into smaller components. Communication guarantees that these concurrent parts will cooperate to determine the correct circuit behavior. For example, communication processors, such as the MAGIC chip [3] and an ethernet coprocessor [4], are representative designs of systems specified at this level of abstraction. These descriptions consist of several protocol handlers that execute concurrently and interact through data transfers and synchronization.

Traditionally, system-level designs have been synthesized by high-level synthesis tools [5], where synthesis is performed by partitioning the circuit description into sequential blocks containing operations, which are scheduled over a discrete time and bound to components [6]. This technique is called *single process synthesis* in [7], since it ignores concurrency and communication in the beginning, thus focusing only on the sequential parts of the design. After the synthesis is performed on each concurrent component, they are combined at the lower levels, i.e. at RTL or logic-level. Note that at this level the results are already suboptimal and harder to optimize.

Single process synthesis imposes severe restrictions on system-level designs. First, since only one sequential component is synthesized at a time, the synthesis tool cannot consider the degrees of freedom available in other concurrent parts of the design. Second, the interface uses a model that does not consider communication. As a result, intricate

relations between a model and its environment cannot be enforced during synthesis. Finally, single process synthesis targets area or delay optimization of each sequential block, which may not yield an optimal design since the design contains concurrent and interacting components. For example, the minimization of the execution time in a concurrent specification requires the minimization of delays over execution paths.

This chapter focuses on modeling, analysis and synthesis of concurrent and communicating systems. In particular:

- We present an algebraic model for concurrent and communicating systems that gives a formal interpretation for system-level descriptions, such that these systems can be abstracted, analyzed and synthesized. This model considers important aspects found in control-dominated specifications, such as concurrency, synchronization and exception handling.
- We present a technique to translate the algebraic model to a symbolic finite state machine. To this finite state machine, we incorporate resource constraints and complex timing constraints, such as global min/max timing constraints.
- We present a technique to statically schedule the operations in basic blocks subject to global design constraints. This technique is based on a 0-1 Integer Linear Programming (ILP) model describing the constraints and degrees of freedom of the design to be synthesized. The 0-1 ILP equations are extracted from the symbolic finite state machine and solved by a Binary Decision Diagram solver.

The outline of this chapter (which is also the outline of the tool we developed) can be seen in Figure 1. In Section 1.2, we present the design issues involved during the synthesis of system-level specifications. Then, in Section 1.3, we describe our model for concurrent control-dominated systems, called *control-flow expressions*. Section 1.4 presents a method to translate a control-flow expression and the system's design constraints into a symbolic finite state machine called a *control-flow finite state machine*. Section 1.5 describes a synthesis method for statically scheduling operations by casting the scheduling problem as a 0-1 ILP instance. Then, we present some examples and concluding remarks.

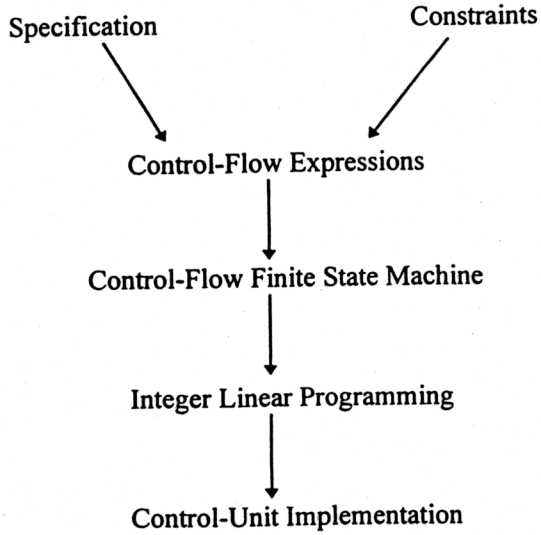

Figure 1: Chapter Outline

1.2. SYSTEM-LEVEL DESIGNS

System-level designs usually contains sub-components showing sequential, alternative, concurrent, repetition and exception handling behaviors [8]. Such systems have been specified in the past with description languages supporting these behaviors, such as VHDL [9], Verilog HDL [10], HardwareC [11], StateCharts [12], and Esterel [13].

Synthesis of system-level designs differs from standard high-level synthesis [7, 14-16] because the emphasis of the tool is placed on concurrent models and their interactions. In addition, implementation of system-level designs are often not confined to a single chip or a hardware implementation alone [17]. As a result, the steps of partitioning, scheduling, synchronization, interface synthesis, and datapath generation in system-level synthesis will focus in the generation of controllers subject to constraints crossing the concurrent models of the specification and different implementation paradigms.

1.2.1. Synthesis Tools used for System-Level Designs

Many systems implemented by Application Specific Integrated Circuits (ASICs) are control-dominated applications [18]. In such applications, high-level synthesis techniques have been used previously to synthesize control-units for system-level designs.

The Olympus Synthesis System [14] targets control-dominated ASIC designs. Starting from the high-level language HardwareC, the system performs the high-level synthesis tasks of scheduling operations over discrete times, binding operations to components and variables to registers. One of the unique features of the Olympus Synthesis System is that it allows the user to specify synchronization and data transfers using high-level message passing communication constructs. In this system, *send* and *receive* operations are used to generate synchronizations and to transfer data across concurrent models. Although HardwareC allows the system to be specified using concurrent and communicating modules, the synthesis technique applied in these modules considers only one module at a time, preventing the synthesis from utilizing the degrees of freedom from the other modules during the synthesis of a single module.

The HIS System [15] was developed at IBM to synthesize mixed dataflow intensive/control-flow intensive specifications. The system being synthesized was first partitioned into its control-flow/dataflow components, for which a control unit and datapath were generated, respectively [19]. In path-based scheduling, operations in a path can be scheduled into a single discrete time as long as it does not have any conflicts with the other operations scheduled in the same discrete time. Because scheduling is performed on a path-basis, this algorithm is able to schedule operations across sequential, alternative and repetitive control-flow structures.

The Princeton University Behavioral Synthesis System [7] (PUBSS) and the Synopsys Behavioral Compiler [16] were conceived using ideas similar to those of the HIS system. Both systems allow control-flow with arbitrary sequential, alternative and repetitive behaviors. In addition to that, PUBSS is able to consider more aggressive timing constraints than the previous systems described in this section, called *path activated constraints*. PUBSS is also able to handle the tightly coupled parts of the design by

merging them together during synthesis. Nevertheless, it is not able to cross parallel composition barriers, which may exist in Verilog or StateChart descriptions.

The Clairvoyant system [20] was designed for the specification and control generation of control-dominated applications using a grammar-based specification language. The system is specified using a grammar languages supporting sequential, alternative and parallel composition, loops, synchronization and exception handling. Since the Clairvoyant system does not allow the incorporation of any design constraints, the synthesis technique is limited to a syntax-directed translation from the grammar specification to the control-unit, and thus all timing information must be already present and scheduled during the specification of the design.

We will describe in this thesis a tool called *Thalia*[1] for system-level synthesis that will be unique because it will be able to handle several of the design issues regarding system-level designs, some of which were mentioned in this section. We will consider specifications containing sequential, alternative, parallel compositions, loops and exception handling mechanisms. Such constructs are present in Verilog, StateCharts, and VHDL. We will not limit the specifications to contain concurrency only at the highest levels of the specification, as it is the case in IBM Synthesis System, PUBSS and Synopsys Behavioral Compiler. We will be also consider general forms of design constraints, which will help us to model the environment.

We will describe now an example whose constraints cannot be automatically incorporated into the design with typical High-Level Synthesis tools.

1.2.1.1. Synchronization of Concurrent Processes of an Ethernet Coprocessor

In this example, we show how we can synchronize multiple processes by statically scheduling operations. We will see that this synchronization can be synthesized only if we consider the degrees of freedom among the different processes that execute concurrently with the model that we want to synthesize.

The block diagram of an ethernet coprocessor is shown in Figure 2. This coprocessor contains three units: an execution unit, a reception unit and a transmission unit. These three units are modeled by thirteen concurrent processes. The problem that we want to solve is the synthesis of the individual controllers subject to the timing constraints imposed by the other concurrent models.

Let us focus on the process *xmit_frame* of Figure 2. This process interacts with two other processes, *dma_xmit* and *xmit_bit*. The process *xmit_frame* was specified as a program state machine written in Verilog HDL, as shown in Figure 3 [4]. In order to synthesize a valid controller, we must observe the constraints imposed by *dma_xmit* and *xmit_bit*.

The process *xmit_frame* works as follows. 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 constrains the rate at which the bytes are transmitted from *xmit_frame*. Thus, any scheduling for the operations in

[1] The muse of comedy

the program states of *xmit_frame* will have to consider the timing constraints crossing basic blocks that are imposed by *xmit_bit*.

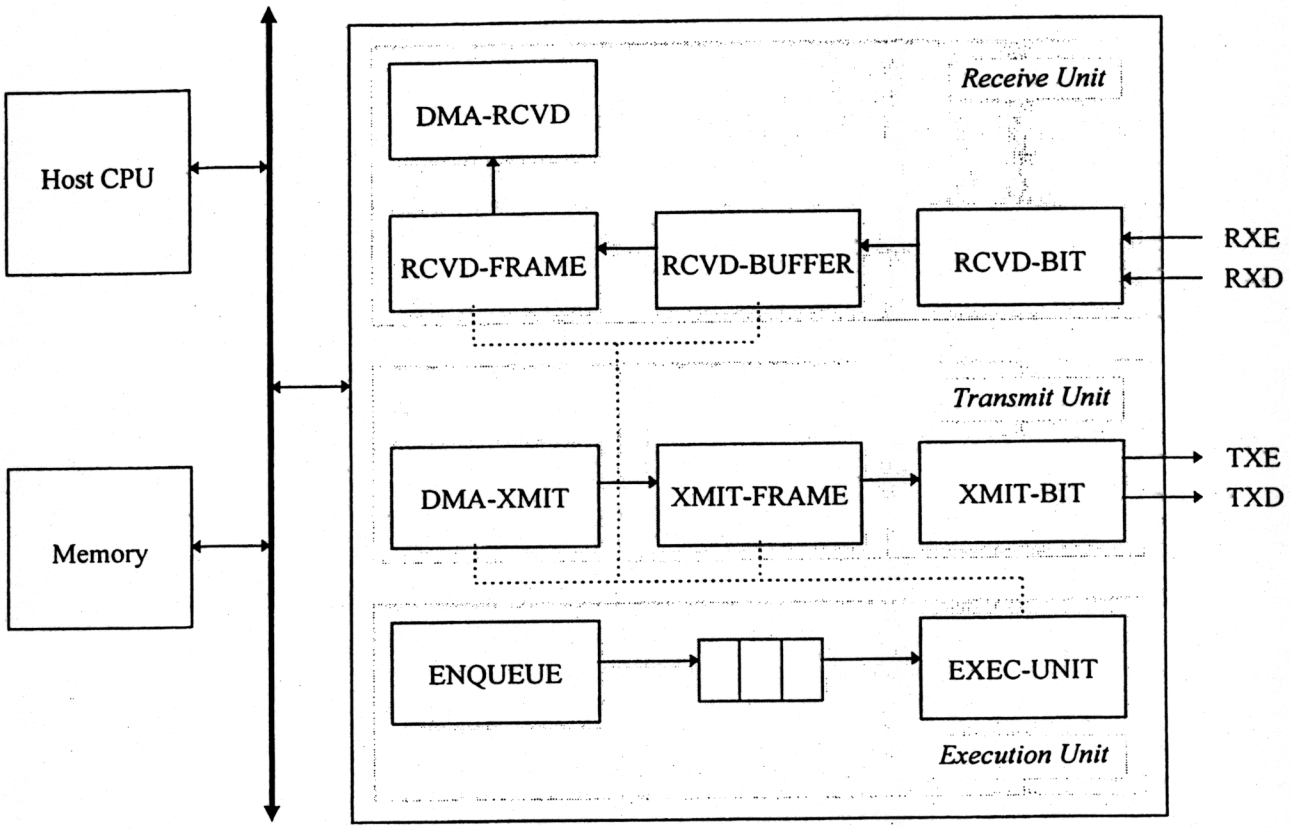

Figure 2: Ethernet Controller Block Diagram

Related Work in Synchronization

Filo et al. [21] addressed the problem of synchronizing operations subject to design constraints imposed by concurrent models by 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 synchronizations due to the transfers are considered during the simplification. A synchronization is eliminated if its execution is guarded by a previous synchronization. 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 achieve the simplification of synchronization that crosses loops and conditionals, and we do not restrict this simplification to only transfers present in single loops or conditional branches, as in [21].

In [22], the system was specified by a set of finite state machines and a set of properties specified using CTL (Computation Tree Logic) formulae. These formulae characterized the desired behavior of the system in terms of safety ("nothing bad ever happens") and liveness ("something good eventually happens") properties. Each machine of the system was considered to execute asynchronously with respect to the other machines, and a product machine was obtained by combining the machines of all specifications. A synchronizer was extracted from the product machine such that this sub-machine satisfied the set of CTL formulae. A similar method was also reported

in [23], but using linear time temporal logic formulae for specifying the temporal properties of the system. This model considered concurrency of the specifications as an interleaving of executions, as opposed to the model we will define in the next section, which will consider true concurrency. As a result, the synchronization generated by these procedures will be subject to much stricter constraints than they will experience.

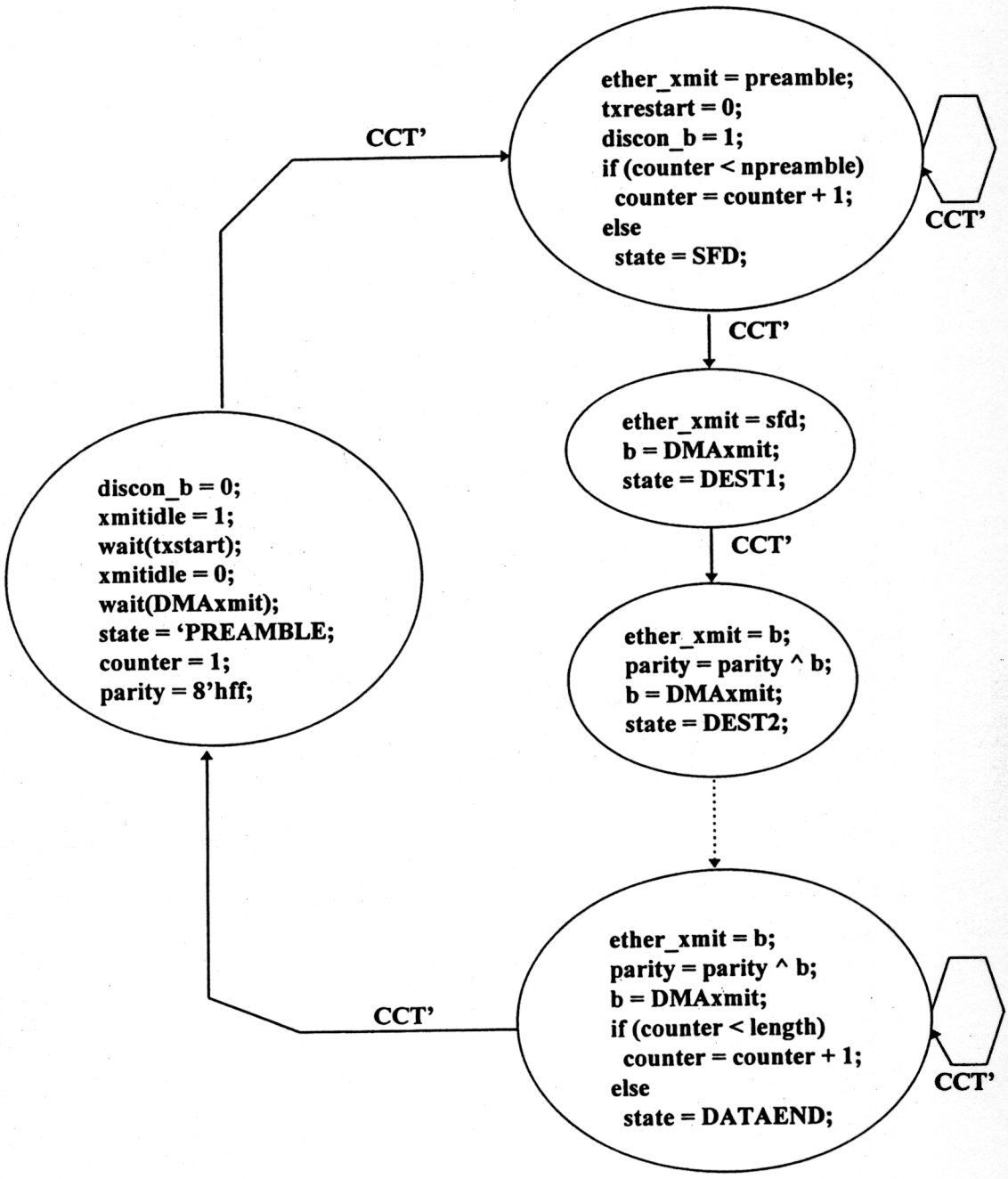

Figure 3: Program State Machine for Process *xmit_frame*

Zhu et al. [24, 25] used timing expressions to capture synchronizations of models. A timing expression is an expression containing timing relations between a set of signals, which are expressed using traces of executions. In his descriptions, the system is specified by a set of timing expressions and the synchronization is specified by a set of

constraints a system has to satisfy. These constraints have been solved by [26] using an algorithm that returns a set of timing expressions for the synchronizers. Timing expressions can be useful for determining relationships among signals in a timing diagram, as shown in [25], when every signal of a timing diagram is represented by a timing expression and the synchronization constraints represents how these signals interact. However, timing expressions will not be able to capture the intricate relations that are present in higher-level descriptions.

In the following sections, we will introduce our algebraic model and the related analysis and synthesis algorithms.

1.3. MODELING OF CONCURRENT SYNCHRONOUS SYSTEMS

We focus in this section on a model for control-dominated system-level descriptions. Since system-level descriptions are usually specified as sets of concurrent components interacting among themselves and with the environment, the correct characterization of the constraints for synthesis can be obtained only if we understand the underlying behavior of the system, and its relation to the environment.

Several models for specifications have been proposed in the past that separates the behaviors in terms of their control-flows and dataflows. We refer the reader to [1, 27, 28] for an introduction to these models. In this chapter, we consider a model of the system in terms of these control-flow and dataflow components, but with the following differences. First, variables and their operations are not confined to the dataflow, since these variables sometimes dictates the control-flow behavior of the system. Second, the control-flow behavior includes exception handling mechanisms and concurrency at any level of the specification hierarchy. Such mechanisms are found in many description languages. Third, we restrict the specification of sets of operations to basic blocks. In the following example, we show how a description language such as the Verilog HDL can be modeled in terms of its control-flow and dataflow components.

Example 1: In Figure 4, we show the representation of a specification in terms of its control-flow and dataflow graphs.

The vertices *loop* and *alt* in the control-flow graph represent iterative and alternative behavior, respectively.

We labeled each operation in the dataflows by events $a_1 \cdots a_6$. Such events are generated by the control-flow and determine when the corresponding operations will execute. Event a_1, for example, triggers the execution of the negation of dx. These events determine the dependency of the dataflow with respect to the control-flow. Each dataflow also contains two vertices, *source* and *sink* that do not correspond to any operation in the specification. They mark the beginning and end of execution of the dataflow, respectively.

The dataflow of Figure 4 generates input events c_1 and c_2 that trigger the execution of the loop and the execution of the alternative path, respectively. These events determine the dependency of the control-flow in terms of the dataflow.

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 event c_1 is generated) and exiting the loop are equally probable, for example.

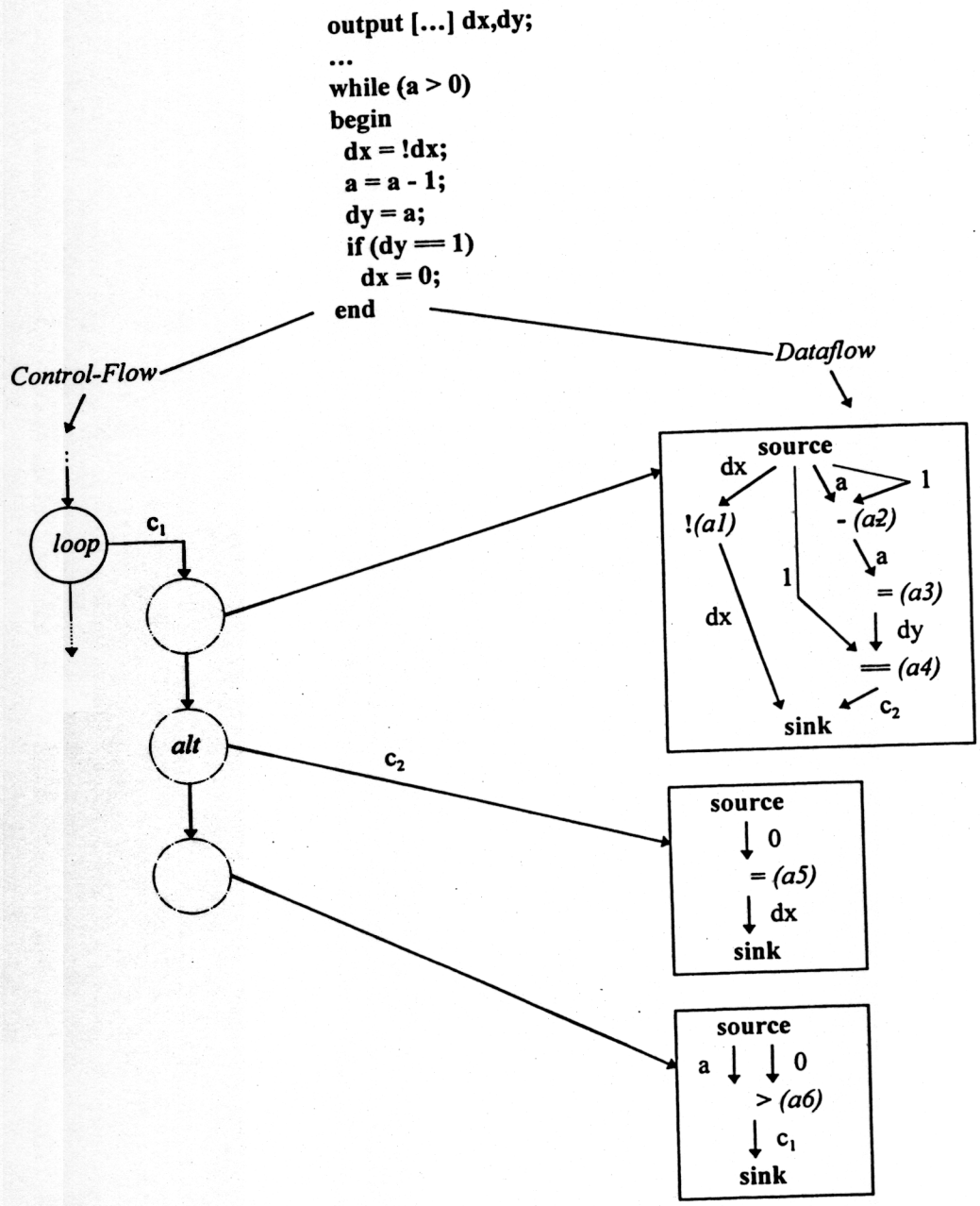

Figure 4: Partitioning of Specification into Control-Flow/Dataflow

In the remainder of this section, we will define our formal control-flow model for system-level descriptions.

1.3.1. Algebra of Control-Flow Expressions

The algebra of control-flow expressions (CFEs) 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 **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: The C fragment presented below corresponds to a part of a differential equation solver found in [1].

```
x1 = x + dx;
u1 = u - (3 * x * u * dx) - (3 * y * dx);
y1 = y + u * dx;
c = x1 < a;
```

During the compilation of this description, the expressions are broken into a set of predefined operations including addition, multiplication, subtraction and comparison.

```
m1 = 3 * x;                    /* m1 */
m2 = u * dx;                   /* m2 */
m3 = m1 * m2;                  /* m3 */
m4 = 3 * y;                    /* m4 */
m5 = m4 * dx;                  /* m5 */
m6 = u * dx;                   /* m6 */
a1 = x + dx;                   /* a1 */
y1 = y + m6;                   /* a2 */
c = a1 < a;                    /* lt */
s1 = u - m3;                   /* s1 */
u1 = s1 - m5;                  /* s2 */
```

If we assume that each operation described above executes in one cycle, we can represent the operations above by actions m_1, m_2, m_3, m_4, m_5, m_6, a_1, a_2, lt, s_1 and s_2, according to the comments to the right of the code.

We represent the input events of a control-flow by *conditionals*, which are symbols from an alphabet **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* $\mathbf{G}$ *to denote the set of guards over conditionals.*

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* $(x \leq y)$ $x = y * z$, a conditional c abstracts the binary relational computation $x \leq y$. If at some instant of time, the *guard* c is *true*, $x = y * z$ is executed. If at some instant of time, the *guard* $\bar{c}$ is *false*, the else branch (which is null in this case) is executed.

Using control-flow expressions, we model systems by a set of operations, dependencies, concurrency and synchronization. We encapsulate sub-behaviors 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 is a mapping from labels of the alphabet **F** to control-flow expressions. As we mentioned earlier, we incorporate some of the register variables into the control-flow representation because they affect more the control-flow behavior of the system than the operations in the dataflow. These variables are represented by the set **R**.

We define Σ as the alphabet of actions, conditionals, processes and registers, i.e., $\Sigma = \mathbf{A} \cup \mathbf{C} \cup \mathbf{F} \cup \mathbf{R}$, where **A** is the set of actions, **C** is the set of conditionals, **R** is the set of register variables with a finite number of possible values and **F** is the set of process variables.

The set of operations is defined as $\mathbf{O} = \{., +, :, *, \omega, \zeta, ||, [:=], [++], [--], \{\mathbf{BB}\}\}$, where . denotes sequential composition, + denotes alternative composition, : denotes guarded execution, * denotes loops, ω denotes unconditional repetition, || denotes concurrency, ζ denotes exception handling, [:=],[++] and [--] are functions defined on registers, and {**BB**} are basic blocks representing sets of operations.

The compositions that are defined in the algebra of control-flow expressions are the compositions supported by existing HDLs which were captured by the control-flow model described earlier. Verilog HDL, for example, supports sequential composition, alternative composition, loops, parallelism, unconditional repetition and exception handling implemented as the disable construct. Similar sets 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 CFEs will also be guarded.

The formal definition of control-flow expressions [28] is presented below.

Definition 3.2: *Let (Σ, O, δ, ε) be the algebra of control-flow expressions where:*

- *Σ is an alphabet that is subdivided into the alphabet of actions, conditionals, registers and processes;*
- *O is the set of composition operators that define sequential, alternative, guard, loop, infinite, parallel behavior, exception handling, basic blocks and operations over registers;*
- *δ is the identity operator for alternative composition;*
- *ε is the identity operator for sequential composition.*

Guards in control-flow expressions are defined as Boolean functions over conditionals, but to these, we include relational operations between registers and constants, such as comparisons.

Definition 3.3: *Control-flow expressions are:*

- *Actions $a \in \mathbf{A}$*
- *Processes $p \in \mathbf{F}$.*
- *δ and ε.*
- *$\zeta(n,p)$, where n is a natural number and $p \in \mathbf{F}$.*
- *$\{r_1, r_2, ..., r_m\}$, where r_i is a precedence relation of the form*
  $a_j \xrightarrow{n} a_k$ $\quad \{a_j, a_k\} \subseteq \mathbf{A}$
- *[v := constant], [v++], [v--], where v is a register.*
- *If $p_1, ..., p_n$ are control-flow expressions, and $c_1, ..., c_n$ are guards, then the following expressions are control-flow expressions.*
  - *The sequential composition, represented by $p_1. ... . p_n$*
  - *The parallel composition, represented by $p_1 || ... || p_n$*
  - *The alternative composition, represented by $c_1:p_1 + ... + c_n:p_n$*
  - *Iteration, represented by $(c_1:p_1)^*$*
  - *Unconditional repetition, represented by p_1^{ω}*

Nothing else is a control-flow expression.

The compositions that are defined in the algebra of control-flow expressions are the compositions supported by existing HDLs which were captured by the control-flow model described earlier. Verilog HDL, for example, supports sequential composition, alternative composition, loops, parallelism, unconditional repetition and exception handling, which is implemented as the disable construct. Similar sets of compositions are also supported in VHDL and HardwareC, and thus are supported by control-flow expressions. Since alternative compositions and loops in these languages are guarded, their corresponding compositions in CFEs will also be guarded.

Informally, we define the behavior of the compositional operators of CFEs as follows: the sequential composition of $p_1, \ldots, p_n$ means that p_{i+1} is executed only after p_i is executed, for $i \in \{1, \ldots, n-1\}$. The parallel composition of $p_1, \ldots, p_n$ means that all p_i's begin execution at the same time for $i \in \{1, \ldots, n-1\}$. The alternative composition of $p_1, \ldots, p_n$ guarded by $c_1, \ldots, c_n$, respectively, means that p_i only begins execution if the corresponding c_i is *true*. Iterative composition means that p_1 begins execution while the guard c_1 is *true*. The infinite composition means that p_1 begins execution infinitely many times upon reset. All possible executions of the basic block $\{r_1, r_2, \ldots, r_m\}$ must observe the basic block's precedence relations r_i. A precedence relation $r_i = a_j \xrightarrow{n} a_k$ defines a minimum execution time of n cycles between a_j and a_k. The register operations $[v := constant]$, $[v{+}{+}]$ and $[v{-}{-}]$ assign to register variable v the value *constant*, increments its value by 1 and decrements its value by 1, respectively. In the case of the exception handling mechanism, we implemented it in a similar way to the Verilog HDL disable construct. Thus, $\zeta(n,p)$ means that we are aborting the execution of the block n levels above ζ in CFE p, the original CFE.

We introduced in the previous definition the symbol δ that is called here deadlock[1]. The symbol δ is defined as $\delta \xrightarrow{\Delta} false{:}p$, 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.

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.

Note that in our definition of the syntax of CFEs, every loop and every alternative branch is guarded by ":", which makes the different branches of alternative and loops distinct. We also assume that only one alternative branch will be taken at any given time. This restricts the specification of loop bodies and alternative branches to only accept deterministic choices with respect to the guards.

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, i.e. we should be able to eliminate recursion from a control-flow expression by substituting process variables by their respective CFE, with the recursion on a process variable being replaced by iterative or

---

[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.

unconditional repetition. In this chapter, whenever we refer to a CFE p, we are referring to the CFE without recursion defined by the process variable p.

In control-flow expressions, 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.

Composition	HL Representation	Control-Flow Expression
Sequential	**begin** *p; q* **end**	$p \,.\, q$
Parallel	**fork** *p; q* **join**	$p \,\|\|\, q$
Alternative	**if** *(c)* *p;* **else** *q;*	$c : p + \bar{c} : q$
Loop	**while** *(c)* *p;* **wait** *(!c)* *p;*	$(c : p)^*$ $(c : 0)^* . p$
Infinite	**always** *p;*	p^{ω}
Exception Handling	**begin** : P_1 ... **disable** P_1; ... **end**	ζ *(level of* P_1, p*)*

Table 1: Link between Verilog HDL Constructs and Control-Flow Expressions

Whenever possible, we will relate the HDL constructs to control-flow expressions, instead of using the control-flow/dataflow model described earlier for sake of simplicity.

The semantics of the major control-flow constructs in HDL are related to control-flow expressions in the table in the Table 1, where p and q are processes ($p,q \in \mathbf{F}$) and c is a conditional ($c \in \mathbf{C}$). In this figure, we relate CFEs to the control-flow structure of Verilog HDL[10]. In this chapter, we assume that guards *(:)* have precedence over all other composition operators; loops and infinite composition $(^*, ^{\omega})$ have precedence over the remaining compositions; sequential composition *(.)* has precedence over alternative and parallel composition; alternative composition *(+)* has precedence over the parallel composition. In addition to that, we use parentheses to overrule this precedence and for ease of understanding. Note that we did not mention basic blocks and register operations in this table. Basic block will be an encapsulation for sets of operations in control-flow expressions. Register operations, on the other hand, will require further considerations that will be discussed later in this section. Although it is not necessary, we will at times replace parentheses by square brackets for clarity.

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 ($\underbrace{p.\cdots.p}_{n}$), 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 $(x{:}p + \bar{x} : \varepsilon)^{n-1}$.

In our original specification, we assumed that every action in **A** takes a unit-time delay in CFEs, 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 $c{:}p + \bar{c} : q$) and executing the first action of process p, the execution of this action would make $\bar{c}$ *true* and thus also enable the execution of q. In order to avoid this erroneous behavior, we adopt a weaker version of the *synchrony hypothesis[13].*

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 order words, c:p can be viewed as (c:ε) . 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.*

```
fork : BLK0
 begin : BLK00
  while (pce) @(posedge clk) out = preamble;        o1
  out = sfd;                                        o2
  out = destination[0];                             o3
  out = destination[1];                             o4
  out = source[0];                                  o5
  out = source[1];                                  o6
  out = length;                                     o7
  i = 0;                                            i1
  while (length > 0)
  begin
   @ (posedge clk) out = data[i];                   o8
   i = i + 1;                                       i2
   length = length - 1;                             l1
  end
  out = eof;                                        o9
  disable BLK0
 end
 begin : BLK01
  wait (posedge CCT);
  disable BLK0;
 end
join
```

Figure 5 : Exception Handling in Verilog HDL

Example 4: The Verilog HDL code of Figure 5 represents a controller that puts a frame in block of data, as in the case of a communications controller. This code contains two concurrent parts, a sequential code and an exception handler.

The first block executes a sequential code which puts the frame on the data block and transmits it to an output port. The second block disables the first one if signal *CCT* becomes true while executing block **BLK00**, indicating that the transmission has been interrupted. Conditional *len* corresponds to the result of the comparison *length* > *0*.

The CFE for this Verilog is presented below, where p_{00} is the CFE for block **BLK00** and p_{01} is the CFE for block **BLK01**.

$$
\begin{aligned}
p &= (p_{00} \,||\, p_{01}) \\
p_{00} &= ((pce{:}o_1)^*.o_2.o_3.o_4.o_5.o_6.o_7.i_1.(len{:}o_8.i_2.l_1)^*.o_9.\zeta(2,p) \\
p_{01} &= ((\overline{CCT}:0)^*.\zeta(2,p)
\end{aligned}
$$

In the remainder of this section, we will discuss the rationale behind the incorporation of basic blocks and register variables in our formalism.

1.3.2. Basic Blocks

We now revisit our definition of CFEs and we show how basic blocks can be efficiently used to encapsulate the behavior of sets of operations.

Definition 3.4: *Let* **A** *be a set of actions, and let the relation* $\rightarrow : \mathbf{A} \times \mathbf{Z} \times \mathbf{A}$ *represent precedence constraint between two actions, where* **Z** *is the set of integer numbers. Then,* $a_1 \xrightarrow{n} a_2$ *corresponds to specifying that action* a_1 *must be executed at least n cycles before action* a_2.

In the definition of a precedence constraint, we will use the shorthand notation $a_1 \rightarrow a_2$ whenever n is 1.

Having defined precedence constraints enables us to define a basic block as one of the possible compositions for control-flow expressions.

Definition 3.5: *Let* r_i *be a precedence constraint. A basic block is an control-flow expression represented by the set of precedence constraints* $\{r_1, r_2, \ldots, r_m\}$.

Note that a basic block encapsulates all the valid implementations for a set of operations, and as a result, it can be represented by the enumeration of a number of alternative paths. Thus, the basic block definition in control-flow expressions only adds a compact and efficient representation for dataflow representations. We will denote a generic basic block by {**BB**}.

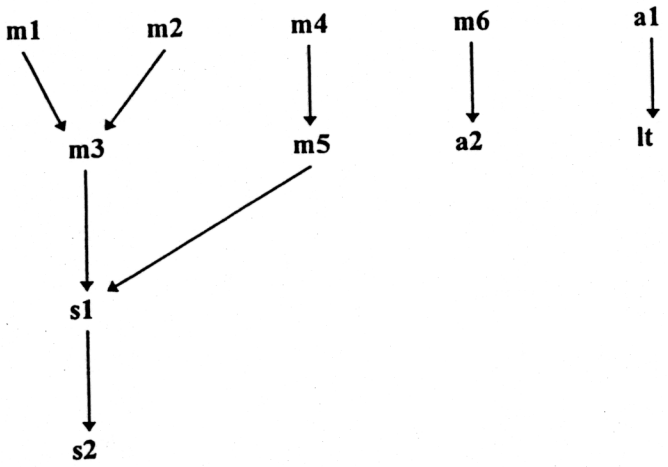

Figure 6: Dataflow for Differential Equation Fragment

Example 5: The dataflow graph of Example 2 is presented in Figure 6. This dataflow can be represented by the control-flow expression

$$\{m_1 \rightarrow m_3,\ m_2 \rightarrow m_3,\ m_4 \rightarrow m_5,\ m_6 \rightarrow \ a_2,\ a_1 \rightarrow lt,\ m_3 \rightarrow s_1,\ s_1 \rightarrow s_2,\ m_5 \rightarrow s_2\}.$$

If the actions corresponding to multiplications executed in two cycles, and all other actions executed in one cycle, then the basic block representing this new set of precedence constraints is given below:

$$\{m_1 \xrightarrow{2} m_3,\ m_2 \xrightarrow{2} m_3,\ m_4 \xrightarrow{2} m_5,\ m_6 \xrightarrow{2} a_2,\ a_1 \rightarrow lt,\ m_3 \xrightarrow{2} s_1,\ s_1 \rightarrow s_2,\ m_5 \xrightarrow{2} s_2\}$$

1.3.3. Register Variables

In order to evaluate the importance of adding register variables to CFEs, let us consider Figure 7. If we adopt the conventional control-flow/dataflow partitioning paradigm, variable *state* is placed into the dataflow. Note, however, that this variable is not connected with any other part of the dataflow, yet it triggers the execution of some parts of a control-flow expression. This means that if we move variable *state* into the control-flow, the communication between the control-flow and dataflow will be reduced. This has some advantages from a synthesis perspective. First, since the *state* variable is now incorporated into the control-flow, the redundancy of control in the dataflow can be eliminated, thus reducing the size of the final implementation. Second, when imposing constraints to the design, we will have a more accurate execution model for the control-flow, which will be more independent on the dataflow abstraction.

Control-flow/dataflow transformations have been regarded in the past as useful transformations [27, 29, 30]. However, only *ad hoc* methods were presented, and it was claimed that these transformations would probably increase the number of states of the control.

We will first define a reduced dependency graph below, whose structure will allow us to determine which variables should be moved to the control-flow.

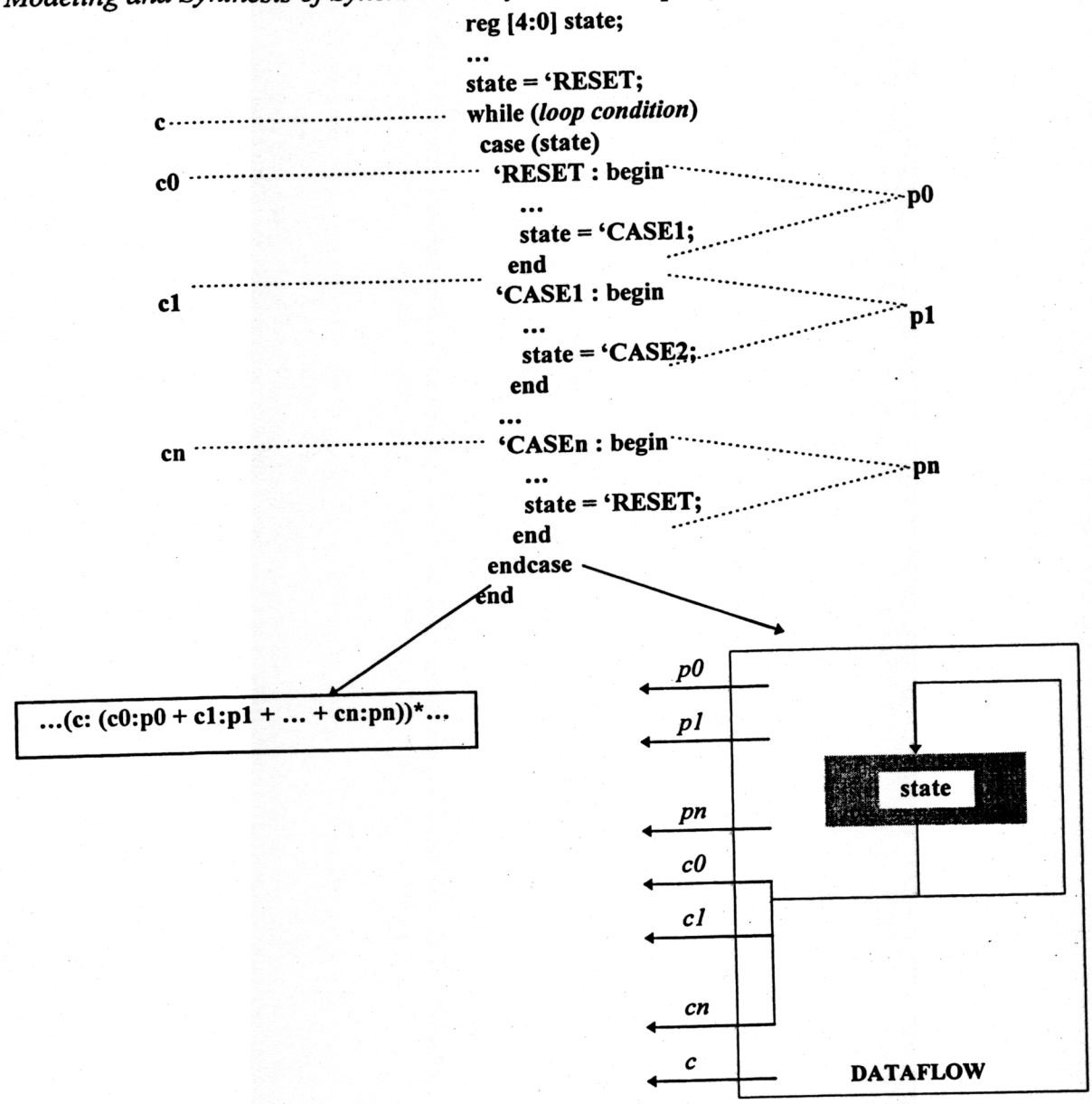

Figure 7: Program-State Machine Specification

Let $\mathbf{D}_f$ be the set of dataflows of a specification.

Definition 3.6: *A reduced dependency graph is the undirected graph* $G_r = (V_r, E_r)$, *where* V_r *is the set of non-constant variables, and an edge between two variables u and v exists if u depends on v or if v depends on u in at least one of the dataflows of* $\mathbf{D}_f$.

In this definition, a reduced dependency graph collapses all the dependencies occurring in the different dataflow graphs, thus disregarding the dependency of the dataflows with respect to the control-flow. Recall that a variable in a dataflow graph can generate events to the control-flow; thus, the reduced dependency graph can be easily annotated with the variables that are used to generate events to the control-flow.

Because of the nature of specifications in programming languages, not all of the vertices in a reduced dependency graph will be connected, i.e., in general, there will be some variables u and v for which no path will exist between u and v. Let $S = \{S_1, ..., S_n\}$ be a partition of the set of vertices V_r such that vertices u and v belong to the same partition if they are connected in G_r.

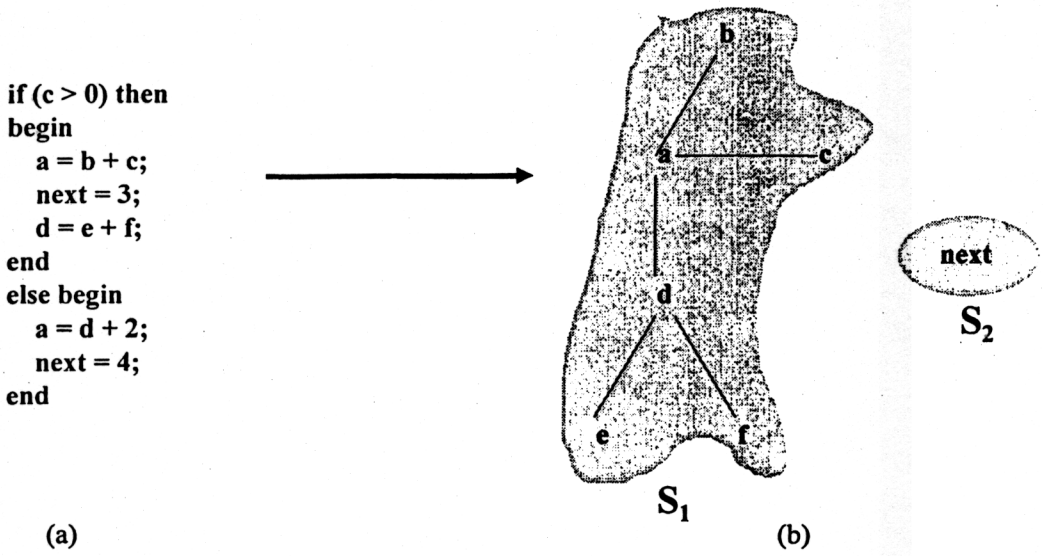

Figure 8: (a) Specification and (b) Reduced Dependency Graph

Example 6: In Figure 8 we present a specification and its reduced dependency graph. The dataflow blocks corresponding to the then and else clauses of the *if* partitions the variables of the specification into two sets, S_1 = {a,b,c,d,e,f} and S_2 = {next}. Note that if we considered the *then* clause of the *if* construct alone, variables {a,b,c} would be disconnected from variables {d,e,f}, because the edge between variables *a* and *d* can be obtained only in the dataflow of the else clause.

What happens when one of the blocks S_i of a partition S is connected to the control-flow, but not connected to remaining part of the dataflow? If this block of variables were moved to the control-flow, the number of edges crossing the control-flow and dataflow boundaries (given by the number of actions and conditionals of the specification) would be reduced, thus giving a better dataflow/control-flow partitioning. By reducing the number of actions and conditionals, we would make the system represented by an CFE to have less interaction with the external world, and as a result, it would be more predictable.

Although in theory we could move all of the dataflow into the control-flow, or vice-versa, in practice this becomes infeasible for two reasons. First, the techniques for analyzing and synthesizing dataflows and control-flows are different, and as a result, optimization techniques would be applied in the wrong places. Second, indiscriminately making everything a control-flow may potentially cause an exponential blow-up in the number of states. Thus, any move from dataflow to control-flow and vice-versa must be performed with caution. For a limited set of operations which uses constant operands, variables can be moved into the control-flow without a large penalty to the complexity of the control-flow. We call such variables control-flow variables, and their corresponding variable blocks (S_i) control-flow blocks.

Let S be a partition on the vertices of G_r, a reduced dependency graph, and let S_i be a block of S such that no vertex $v \in S_i$ corresponds to an I/O port of the specification. Then, we can say that S_i is useless or it is a control-flow block.

The basic idea relies on the fact that S_i is disconnected from the remaining part of the control-flow. Thus, if S_i is not connected to the control-flow, it will be useless, since all

the values assigned to its variables will not be used anywhere. On the other hand, if this block of variables is connected to the control-flow, then it will be a control-flow block.

In the sequel we denote by $\sigma = \{v, c_1, ..., c_m\}$ a generic connected component of V_r when c_i are Boolean variables. We also denote by $\mathbf{R} = \{=, \neq, <, >, \leq, \geq\}$ the set of relational operations and by ρ a generic element of $\mathbf{R}$. We also denote by γ a constant. The following corollary is used in our extension to control-flow expressions.

Corollary 3.1: *Let v and c be variables of the connected component Σ. Let also f be either an identity function, an increment or a decrement, and let* $c_j \leftarrow \rho(v_1, ..., v_n)$ *and* $v \leftarrow f(v)$ *be the only operations of the specification defined over* c_j *and v. Then, either* S_i *is a control-flow block or it is useless.*

It remains to be seen that such transformations are useful by showing that these types of specifications occur in real designs. It is not hard to see that the variable *state* from Figure 7 satisfies the conditions of Corollary 3.1. We present in Figure 9 (a) the different dataflows for the description of Figure 7, and in Figure 9 (b) the reduced dependency graph for these dataflows. Other variables that often occur in the specifications of control-dominated specifications are counters, for example.

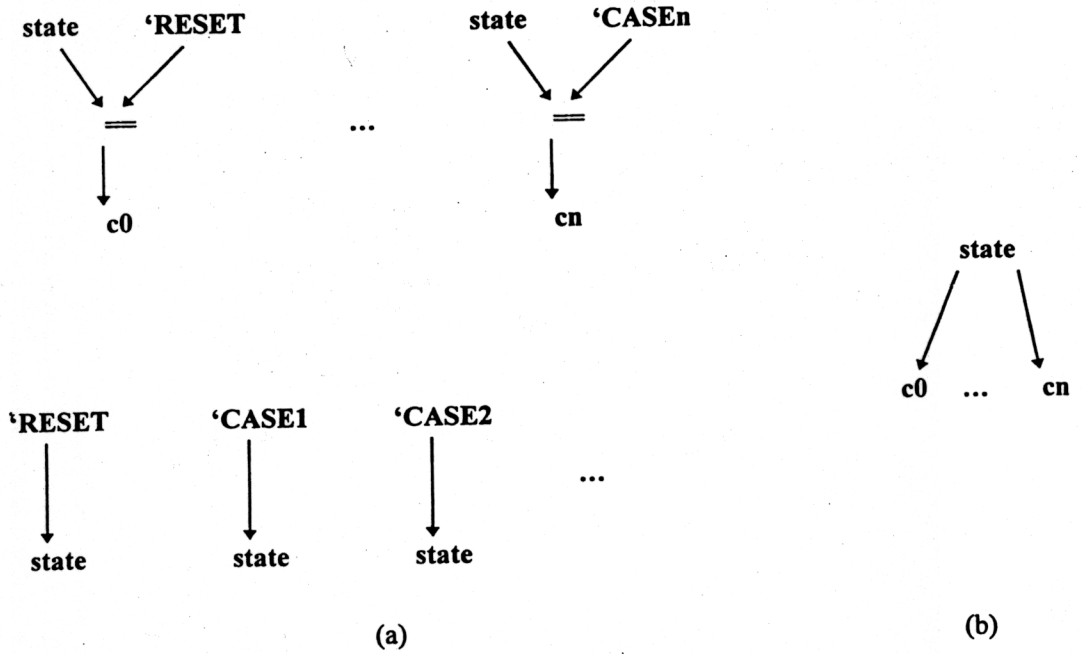

Figure 9: (a) Dataflow Graphs for Program-State Machine and (b) Reduced Dependency Graph

The observations shown in this section led to the definition of register variables and register operations in control-flow expressions. Note that we only incorporated assignment to constants, increments and decrements. We could have incorporated more of the dataflow operations into the control-flow. However, introducing more variables into the control-flow expressions could easily increase complexity in the internal representation of the control-flow.

Note also that every register variable $v \in V_r$ is finite, since the corresponding specification has finite memory in the number of variables. As a result, every operation

performed on the register variable v will be computed over the range $\{0,...,|v|-1\}$, where $|v|$ is the number of possible values for the register.

```
(out,states,disable) = cfe2cffsm(in, kill, p) {
 kill = kill ∨ p.disable();
switch (p.type ()) {
 case ACTION :
   out = create_new_register();
   out.guard() = in ∧ kill';
   Af(p.action()) = Af(p.action()) ∨ out;
   return (out,out);

 case DISABLE :
   p.block().disable() = p.block().disable()  ∨ in;
   return (0,0);

 case SEQUENTIAL :
   states = empty_set();
   foreach (pi in p1 . ... . pn) {
     (out[i],states[i]) = cfe2cffsm (in, kill, pi);
     in = out[i] ∨ pi.disable();
     states.union(states[i]);
   }
   return (out[n] ∨ p.disable(), states);

 case PARALLEL :
   states = empty_set();
   out = true;
   foreach (pi in p1 || ... || pn) {
     (out[i],states[i]) = cfe2cffsm (in, kill, pi);
     out[i] = wait_all_branches (p, out[i]);
     out = out ∧ out[i];
     states.union(states[i]);
   }
   return (out, states);

 case ALTERNATIVE :
   states = empty_set();
   out = false;
   foreach (pi in c1 : p1 + ... + cn : pn) {
     (out[i],states[i]) = cfe2cffsm (in ∧ ci, kill, pi);
     out = out[i] ∨ out[i] ∨ pi.disable();
     states.union(states[i]);
   }
   return (out, states);

 case INFINITE :
   net = create_new_net (in);
   (out, states) = cfe2cffsm (net, kill, pi);
   net = net ∨ out;
   return (0, states);

 case LOOP :
   net = create_new_net (in ∧ ci);
   (out, states) = cfe2cffsm (net, kill, pi);
   net = net ∨ out ∧ ci;
   return ((in ∨ out) ∧ ci', states);

 case BASIC_BLOCK :
   states = empty_set ();
   for(i=1; i≤ALAP; i++) {
     out = create_new_register ();
     out.guard = in ∧ kill' ∧ Fi;
     in = out;
     states.union (out);
   }
   return (out, states);
 }
}
```

Figure 10: Procedure for the Translation of a CFE into a CFFSM

1.4. SEMANTICS OF CFES

In this section, we present the semantics of CFEs by translating it to a Control-Flow Finite State Machine (CFFSM).

A CFFSM is defined as the tuple (I,O,Q,δ,A_f,q_0) *[31]*, where I is the set of input variables, O is the set of output variables, Q is the set of states, δ is the transition function $(2^I \times Q \rightarrow Q)$, A_f is the output function $(O \rightarrow 2^Q)$, and q_0 is the initial state.

The CFFSM is related to the CFE in the following way. The set of inputs of the CFFSM *(I)* corresponds to the conditional variables, and the set of outputs of the CFFSM *(O)* corresponds to the actions. The states of CFFSMs are created for each occurrence of actions and register operations in the CFEs. Due to this similarity, we show in this section the procedure considering only actions, since register operations can be obtained in a similar manner, but with the addition of the corresponding registers. In the CFFSMs, we also call A_f an *action activation function*, since it determines when an action from the CFE can be executed.

We use a procedure *(out,states) = cfe2cffsm(in,kill,p)*, that upon receiving an input *in* and *kill* signal as the execution guards for CFE p, computes the CFFSM for p and returns a condition (*out*) representing when the CFFSM exits the execution of p, and a collection of the new states for the CFFSM (*states*). We assume that each block contains a disable mechanism (*p.disable()*) that collects all the input guards of a disable command to that block. The mechanism *p.disable()* tells us when the block is forced to exit due to a $\zeta(n,p)$.

In basic blocks, we generate a state for each possible execution time of the basic block, given by its as soon as possible and as late as possible schedules. Then, for each action i that can be scheduled in the basic block, we create a Boolean decision variable x_{ij} whose value will be determined during synthesis. Whenever x_{ij} is 1, action i is scheduled to be executed in state j. We assume here single cycle actions, although this procedure could be easily extended to include multi-cycle actions [28]. A transition from a state j to state $j+1$ in the CFFSM of a basic block can occur only if some action can be scheduled after j, which is captured by $F_j = \bigvee_{k>j} x_{ik}$. Figure 10 presents the basic algorithm.

1.5. SCHEDULING OPERATIONS IN CFFSM

In this section, we show how to schedule operations from a CFFSM satisfying design constraints while optimizing some design goal. Since we encoded the possible control-unit implementations by decision variables in the previous section, design constraints will be translated into constraints on the possible values these decision variables may have, and a feasible schedule will be determined by assigning values to the decision variables.

We model the scheduling problem as an instance of Integer Linear Programming, which can be represented by following set of equations [1, 27, 32, 33].

$$\begin{aligned} \textbf{min} \quad & \sum_i c_i x_i \\ & A\,x = b \\ & x_i \in \{0,1\} \end{aligned}$$

The solution to an ILP problem is an assignment to variables x_i such that they satisfy the set of constraints $A\,x = b$, while minimizing the cost function $\sum_i c_i x_i$. Here, we are interested in the formulations in which x_i are binary variables, i.e., they can take 0 or 1 values. This problem has been also referred in the literature as a 0-1 Integer Linear Programming problem.

The schedule of operations in basic blocks has been modeled as an ILP instance in the following way. Let us assume that the maximum execution time for a basic block has been fixed. This will impose constraints on the possible scheduling times for the operations of the basic block. For each operation o_i and possible scheduling time j for o_i, let x_{ij} be a Boolean variable such that if x_{ij} has value 1, then operation o_i is scheduled at time j, or equivalently, if $x_{ij} = 1$, then $t_i = j$. We denote each possible execution time between the first and the last cycles of the basic blocks *control-steps*.

We assume that all schedules for an operation inside a basic block are static. Thus, exactly one of x_{ij} for all j's will have value 1. This can be characterized by the constraint shown below.

$$\sum_{j} x_{ij} = 1 \tag{1}$$

For every precedence constraint $o_{i1} \xrightarrow{l} o_{i2}$ in a basic block, the schedules of $i1$ and $i2$ are constrained by the equation $t_{i2} - t_{i1} \geq l$ shown before. Since the execution time of an operation o_i is completely determined by the control-step the operation executes in a basic block, and since both operations are in the same basic block, this precedence constraint can be rewritten as the following inequality.

$$\sum_{j} j\, x_{i2,j} - \sum_{j} j\, x_{i1,j} \geq l \tag{2}$$

Note that timing constraints between two operations in a basic block can be represented similarly. The precedence constraint $o_{i1} \xrightarrow{l} o_{i2}$ already represents the minimum time between o_{i1} and o_{i2} to be l. Maximum time constraints can be obtained by noting what happens when you multiply Inequality 2 by -1, which can be represented by the precedence constraint $o_{i2} \xrightarrow{-l} o_{i1}$[1, 33].

The last type of constraint represent resource bounds. We assume that for each operation type $type(o_i) = k$, there is an associated limit on the number of resources M_k that can be concurrently executing with o_i in the basic block. Thus, this constraint can be represented by the following equation.

$$\sum_{i \text{ such that } type(O_i) = k} x_{ij} \leq M_k \tag{3}$$

Example 7 We present below the Boolean variables defining the schedule for all operations of the basic block of the differential equation of Figure 6, according to a maximum execution time of 4 cycles. Operations m_1, m_2, m_3, s_1 and s_2 only require one variable, while operations m_4 and m_5 require two variables.

$$\begin{aligned}
x_{m1,1} &= 1 \\
x_{m2,1} &= 1 \\
x_{m3,2} &= 1 \\
x_{m4,1} + x_{m4,2} &= 1 \\
x_{m5,2} + x_{m5,3} &= 1 \\
x_{s1,3} &= 1 \\
x_{s2,4} &= 1 \\
2\, x_{m3,2} - x_{m1,1} &\geq 1 \\
2\, x_{m3,2} - x_{m2,1} &\geq 1 \\
3\, x_{s1,3} - 2\, x_{m3,2} &\geq 1 \\
4\, x_{s2,4} - 3\, x_{s1,3} &\geq 1 \\
(2\, x_{m5,2} + 3\, x_{m5,3}) - (1\, x_{m4,1} + 2\, x_{m4,2}) &\geq 1 \\
4\, x_{s2,4} - (2\, x_{m5,2} + 3\, x_{m5,3}) &\geq 1
\end{aligned}$$

In addition to these constraints, if we restrict the number of multipliers to 2, then we obtain the following additional constraints.

$$x_{m1,1} + x_{m2,1} + x_{m4,1} \leq 2$$
$$x_{m3,2} + x_{m4,2} + x_{m5,2} \leq 2$$
$$x_{m5,3} \leq 2$$

The Integer Linear Programming formulation presented above presumes the existence of an objective goal that needs to be minimized. In the scheduling problem, the minimization of the execution time of the basic block and the minimization of some resource usage costs have been used in the past.

The minimization of execution time can be represented by computing the execution time of the last operation of the basic block. Let o_i be the last operation of the basic block, or a *sink* vertex for a basic block, if the basic block does not have a single last operation, and let its possible schedules be determined by i_{min} and i_{max}. Then, the cost function $\sum_{j \in [i_{min}, i_{max}]} j \; x_{ij}$ represents the cost function that characterizes the execution time of the basic block, since the *sink* vertex of the basic block is the last operation that is executed.

We can also obtain an objective cost function that minimizes a resource cost. For each resource type k, let c_k be its cost. Then the cost function $\sum_k c_k M_k$ denotes the cost of the basic block in terms of its resources. Note that in this case, M_k is not assumed to be a constant value, as presented in Inequality 3, but a variable that may take any integer value.

Example 8: For the set of equations presented in Example 1, the minimization of the execution time in the basic block is represented by a cost function that computes when operation s_2 executes. Thus, the cost function is ***min*** $4 x_{s2,4}$.

If the objective goal is the minimization of resource cost, we replace the last 3 equations of Example 7 by the following equations.

$$x_{m1,1} + x_{m2,1} + x_{m4,1} \leq M_m$$
$$x_{m3,2} + x_{m4,2} + x_{m5,2} \leq M_m$$
$$x_{m5,3} \leq M_m$$

In this case, M_m is a variable taking integer values. If we assume that the cost of a multiplier is c_m, the cost function can be represented by ***min*** $c_m M_m$.

1.5.1. Static Scheduling under Complex Timing Constraints

One of the problems with the scheduling formulation presented in the previous section is that they can only solve the scheduling problem for basic blocks, and that cost functions, timing and resource constraints can only be applied to basic blocks. In this section, we present a methodology for incorporating design constraints and applying cost functions to the CFFSM, such that an optimal solution can be found that statically satisfies the design constraints, over a number of basic blocks simultaneously.

Recently, [34] proposed a methodology of Behavior Finite State Machines (BFSMs) for representing sequential and conditional basic blocks (which are called behavioral states). In [35], an algorithm was presented for the scheduling operation in BFSMs that allowed the satisfaction of timing constraints that crossed behavioral states. However,

these techniques were restricted to sequential and conditional blocks, and constraints were limited to path-activated timing constraints.

The formulation for the scheduling problem presented in this section considers not only sequential and conditional paths, as in the case of BFSMs, but also concurrent blocks. In addition to that, we allow the incorporation of resource constraints, and the specification of environment processes. Our objective is the derivation of Integer Linear Programming constraints from the CFFSM, their solution and application back to the CFFSM.

Because in static scheduling the constraints have to satisfy all the execution conditions of the system modeled by the CFE, we will not consider them to be part of the system modeled by the CFE, but we will extract static scheduling conditions from the CFFSM instead, and represent them separately.

1.5.1.1. Extracting Constraints from the CFFSM

We present in this section how we can represent static scheduling constraints of a CFE. Let $M=(Q,I,O,\delta,\lambda,q_0)$ be a CFFSM corresponding to a CFE p. Note that a finite state machine such as the CFFSM M can be represented by a transition relation [36]. Let T be the transition relation of M, and let $\mathbf{D_f}$ be the set of basic blocks of p. For each basic block $d \in \mathbf{D_f}$, we assume the actions $A_d = \{a_1,\ldots,a_n\}$ are the actions defined in the precedence constraints of d, and $A' = \bigcup_{d\in\mathbf{D}_f} A_d$ is the set of actions defined in all basic blocks of p.

Recall that in the scheduling problem presented in the beginning of this section, we defined three types of constraints for the Integer Linear Programming formulation. Equation 1 required that only one schedule for each operation was allowed. Inequality 2 defined the precedence constraints between two operations. Inequality 3 defined resource usage constraints inside a basic block.

In a CFE and its corresponding CFFSM, Equations 1 and 2 can be obtained directly from the precedence constraints of a basic block, and the possible scheduling times for the actions.

Let $a_i \in A'$ be an action, let j range over the possible scheduling times for a_i, and let x_{ij} be a decision variable defined for a_i. Recall that we represented the CFFSM by a transition relation T previously. We also used a efficient encoding $e(x_{ij})$ for the decision variables x_{ij}. For example, if an action a_i could only be executed in the first or second control-steps of a basic block, which corresponds to defining decision variables x_{i1} and x_{i2}, respectively, then a suitable encoding for the decision variables of a_i would be $e(x_{i1}) = \overline{x_i}$ and $e(x_{i2}) = x_i$, where x_i is a Boolean variable.

Since obtaining only one schedule for an action a_i is equivalent to allowing only one of the encodings for x_{ij} to be valid, we can modify Equation 1 to the equation below, where e overloads the encoding function $e(x_{ij})$ and an arithmetic function whose weight is 1 if the encoding $e(x_{ij})$ is evaluated to *true*, and 0 otherwise.

$$\sum_j e(x_{ij}) = 1 \qquad (4)$$

Example 9: Suppose an operation o_1 can be scheduled in cycles *1* and *2*, resulting in the decision variables x_{11} and x_{12}. As discussed earlier, each decision variable will have a corresponding encoding $e(x_{11})$ and $e(x_{12})$. Assume $e(x_{11}) = x_a$ and $e(x_{12}) = x_b$. Then the constraint $e(x_{11}) + e(x_{12}) = 1$ can be rewritten as the arithmetic formula.

$$(1\ x_a + 0\ \overline{x_a}) + (1\ x_b + 0\ \overline{x_b}) = 1$$

It should be clear that this arithmetic function has value 1 if either $x_a\ \overline{x_b}$ or $\overline{x_a}\ x_b$.

If $a_{i1} \xrightarrow{l} a_{i2}$ is a precedence constraint of a basic block, we can only allow the assignments to the corresponding decision variables of a_{i1} and a_{i2} ($x_{i1,j}$ and $x_{i2,j}$, respectively) such that $t_{i2} - t_{i1} \geq l$, which can be represented in a form similar to Inequality 2.

$$\sum_j j\ e(x_{i2,j}) - \sum_j j\ e(x_{i1,j}) \geq l \tag{5}$$

Note that the function e in these set of ILP constraints acts as a linear transformation ($e(c_1 f(x) + c_2 g(x)) = c_1 e(f(x)) + c_2 e(g(x))$) since it is a bijective function and it distributes over arithmetic addition and multiplication by constants. Consequently, Inequality 3 could be easily rewritten as:

$$\sum_{i \text{ such that } type(O_i) = k} e(x_{ij}) \leq M_k \tag{6}$$

When the encoding function e is applied to the set of Inequalities 1, 2, 3, we can no longer use conventional ILP solvers, because the equations are no longer linear in terms of the new Boolean variables. We show later in this section that BDDs can be used to efficiently solve these set of equations.

We present next how we can generalize constraints if we extract them directly from the CFFSM.

Path-Activated Constraints With the CFFSM and the encodings shown previously, we can enforce more complex timing constraints than min/max timing constraints. In [35], Yen called these constraints *path-activated constraints*, since they were imposed over a whole path, instead of just end points.

A path-activated constraint is defined as ***type***$(n,[l_1, \ldots, l_m])$, where **type** is one of **min**, **max** or **delay**, and the term l_i is either a set of actions or a Boolean guard defined over conditionals or comparisons on registers. We present in this section how we can constrain the decision variables of basic blocks in terms of these path-activated constraints.

In order to constrain the decision variables of a CFE from a path-activated constraint, we have to identify how many cycles occur in the CFFSM between two consecutive terms l_i and l_{i+1}. Recall that if l_i and l_{i+1} are actions, then the action activation functions $A_f(l_i)$ and $A_f(l_{i+1})$ will determine in which transitions of the CFFSM the actions l_i and l_{i+1} occur, respectively. Thus, we can constrain the decision variables by counting the number of cycles when we traverse the CFFSM from $A_f(l_i)$ to $A_f(l_{i+1})$, while keeping the decision variables in the traversal.

Since the terms of path-activated constraints include sets of actions and Boolean guards, we have to extend the action activation function A_f for these elements. We call here such

extension an activation function, and it will be denoted by A^*_f. When applied to a set of actions, A^*_f will uniquely identify each action activation function. When applied to Boolean guards, however, A^*_f indicates in which transitions of the CFFSM the Boolean guard occurs. In this case, the activation function consists of two parts, the Boolean guard itself, and the intersection of the transitions for each conditional or register specified in the Boolean formula of l_i. We assume in the following definition that the function C_f: $G \rightarrow Q \times 2^I$ returns the set of transitions in which the Boolean formula guards the transition of the corresponding CFE.

Definition 5.1:

$$A^*_f(l_i) = \bigvee_{a_j \in l_i} x_j A_f(a_j) \qquad \text{if } l_i \text{ is set of actions } \{...,a_j,...\}$$
$$= A_f(a) \qquad \text{if } l_i \text{ is action } a$$
$$= g\, C_f(g) \qquad \text{if } l_i \text{ is Boolean guard } g$$

Note that in the case where $l_i = \{...,a_j,...\}$, we created a new Boolean variable x_j for each action a_j. We call the set of Boolean variables x_j by B. This variable allows us to uniquely identify an action activation function in A^*_f.

Example 10:

Let $p = (\{a_1 \rightarrow a_2, a_1 \rightarrow a_3, a_2 \rightarrow a_4, a_3 \rightarrow a_4\}.\ (c: \{b_1 \rightarrow b_2, b_1 \rightarrow b_3, b_2 \rightarrow b_4, b_3 \rightarrow b_4\})^*)^\omega$, represented graphically in Figure 11 (a), and let both basic blocks to execute in at most 4 cycles. The CFFSM is presented in Figure 11 (b), where et_{a4} corresponds to the condition when the first basic block requires 4 cycles to execute, and et_{b4} corresponds to the condition when the second basic block requires 4 cycles to execute.

Because operations a_4 and b_4 can only execute in the third or fourth cycles of their respective basic blocks, we can consider the exit conditions for the basic blocks in the first, second and third cycles to be always false, since all operations of the basic block must execute, according to our definition of an implementation for a CFFSM. In addition to that, the exit condition for the fourth cycle is always true, because after executing the fourth cycle of the basic block, the basic block must exit.

Note also that a_1 and b_1 can execute in the first or second cycles of their respective basic blocks, a_2, a_3, b_2 and b_3 can execute in the second or third cycles of their respective basic blocks and a_4 and b_4 can execute in the third or fourth cycles of their respective basic blocks.

The Boolean formulae defining the execution time for the basic blocks are presented below, where y_{a4} and y_{b4} represent decision variables created for the fourth cycles of the first and second basic blocks, respectively.

$et_{a4} = (e(x_{a4,4}) \vee e(y_{a4}))$
$et_{b4} = (e(x_{b4,4}) \vee e(y_{b4}))$

The following are activation functions for the CFFSM presented in the figure.

$A^*_f(a_1) = e(x_{a1,1})\ (S_0\ \overline{S_1}\ \overline{S_2}) \vee e(x_{a1,2})\ (S_0\ S_1\ \overline{S_2})$
$A^*_f(c) = c\ (\overline{S_0}\ S_1\ \overline{S_2}\ \overline{et_{a4}} \vee \overline{S_0}\ S_1 S_2 \vee \overline{S_3}\ S_4\ \overline{S_5}\ \overline{et_{b4}} \vee \overline{S_3}\ S_4 S_5)$

The extraction of a path-activated constraint can be easily explained now. The idea is that we traverse the CFFSM represented by a transition relation T starting at $A^*_f(l_1)$, then waiting until $A^*_f(l_2)$ occurs in the traversal, then waiting for $A^*_f(l_3)$, and proceeding until we reach $A^*_f(l_m)$. During the traversal of the reachable states, instead of existentially quantifying all inputs of the CFFSM, we keep the decision variables such that at the

end we have the set of valid assignments for the decision variables. The traversal of the path-activated constraint can be represented by the finite-state machine of Figure 12. We traverse the CFFSM and the machine represented in Figure 12 until we reach state S_F. If the path constraint is ***min***$(n,[l_1,\ldots, l_m])$, then we only keep the assignments to the decision variables for which S_F is reached in more than $n-1$ cycles. If the path constraint is ***max***$(n,[l_1,\ldots, l_m])$, then we only keep the assignments to the decision variables if S_F is reached during the traversal of the CFFSM in less than $n+1$ cycles. If the path constraint is ***delay***$(n,[l_1, \ldots, l_m])$, then we only keep the assignments to the decision variables in which S_F is reached during the traversal in exactly n cycles.

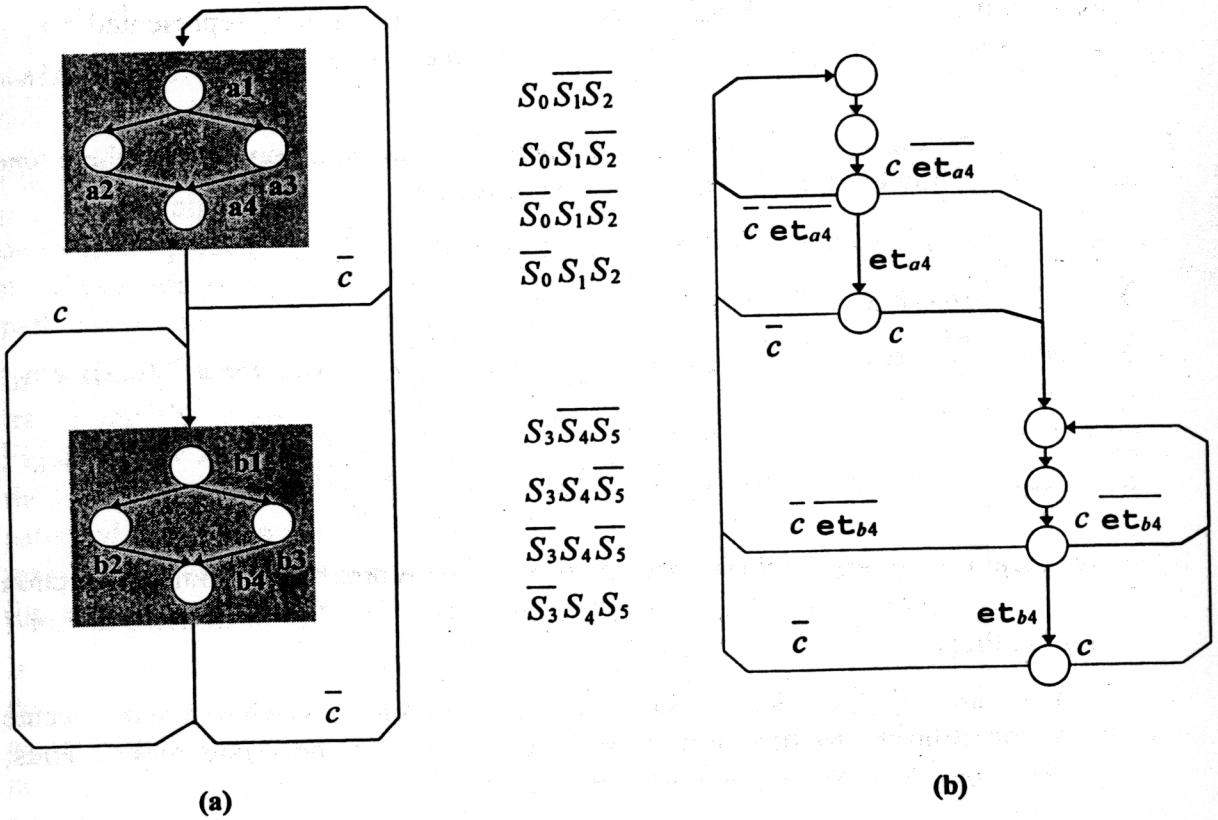

Figure 11: (a) Graphical Representation of CFE p and (b) CFFSM for p

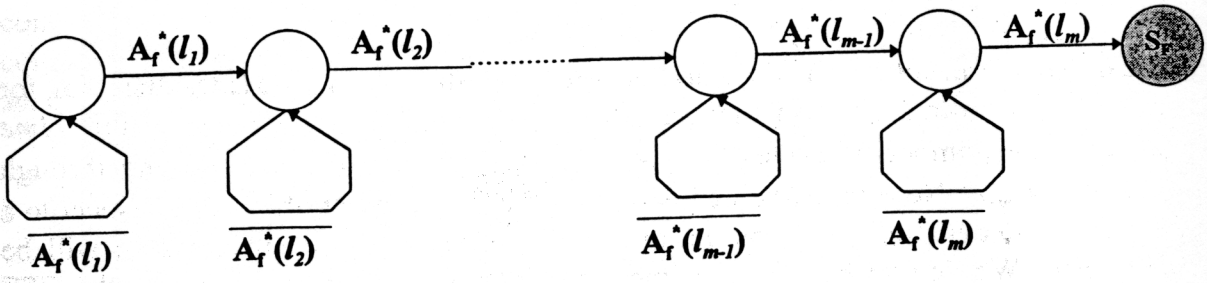

Figure 12: Finite-State Machine Representing the Path-Activated Constraint

Before we present the algorithms for computing the minimum and maximum path-activated constraints, let us show first that this procedure is equivalent to Inequality 5 when two actions are specified in the same basic block.

Theorem 5.1: *If a_1 and a_2 belong to the same basic block, then*

$$\min(n,[a_1, a_2]) = \sum_j j\,x_{2j} - \sum_j j\,x_{1j} \geq n.$$

Proof: First note that $A^*_f(a_1) = \bigvee_j e(x_{1j})\, F^1_j(c,S)$, for some function $F^1_j(c,S)$. Thus, we can consider in this representation of $A^*_f(a_1)$ that $F^1_j(c,S)$ carries the information about the execution time for a_1, while $e(x_{1j})$ carries the decision on whether a_1 will be executed at transition $F^1_j(c,S)$ or not. Since $\sum_j e(x_{1j}) = 1$, then $\bigvee_j e(x_{1j})\, F^1_j(c,S) = \sum_j e(x_{1j})\, F^1_j(c,S)$, the same being valid for $A^*_f(a_2)$. Note also that the product of $x_{1,j1}$ and $x_{2,j2}$ can be replaced by the Boolean conjunction of $x_{1,j1}$ and $x_{2,j2}$, since the only possible values for these variables are 0 and 1. Finally, because $x_{1,j1}$ and $x_{2,j2}$ can be represented $e(x_{1,j1})$ and $e(x_{2,j2})$ respectively, $x_{1,j1}\, x_{2,j2}$ can be replaced by the Boolean conjunction of $e(x_{1,j1})$ and $e(x_{2,j2})$.

The execution time for $t_{a2} - t_{a1} \geq n$ can be represented by the equation below, where *time* is a function returning the time when the action is executed.

$$\Leftrightarrow time(\sum_j e(x_{2j})\, F^2_j(c,S)) - time(\sum_j e(x_{1j})\, F^1_j(c,S)) \geq n$$
$$\Leftrightarrow \sum_j e(x_{2j})\, time(F^2_j(c,S)) - \sum_j e(x_{1j})\, time(F^1_j(c,S)) \geq n$$
$$\Leftrightarrow \sum_j e(x_{1j}) \sum_k e(x_{2k})\, time(F^2_k(c,S)) - \sum_k e(x_{2k}) \sum_j e(x_{1j})\, time(F^1_j(c,S)) \geq n$$
$$\Leftrightarrow \sum_j \sum_k e(x_{1j})\, e(x_{2k})\, (time(F^2_k(c,S)) - time(F^1_j(c,S))) \geq n$$
$$\Leftrightarrow \sum_j \sum_k e(x_{1j})\, e(x_{2k})\, [time(F^2_k(c,S)) - time(F^1_j(c,S)) \geq n] = 1$$

In the last equation, $[time(F^2_k(c,S)) - time(F^1_j(c,S)) \geq n]$ represents a Boolean function that returns 1 if we can traverse the CFFSM from $F^1_j(c,S)$ to $F^2_k(c,S)$ in more than $n-1$ cycles, and 0 otherwise.

Since a_2 and a_1 are both in the same basic block, then the time in which a_1 and a_2 execute will always be relative to the beginning of execution of the basic block. Thus, $time(F^2_k(c,S)) - time(F^1_j(c,S))$ can be replaced by $k - j$.

$$\Leftrightarrow \quad \sum_j \sum_k e(x_{1j})\, e(x_{2k})\, [k - j \geq n] = 1 \Rightarrow$$
$$\Leftrightarrow \quad \sum_j j\, e(x_{2j}) - \sum_j j\, e(x_{1j}) \geq n$$

We can now present a corollary to the theorem above that provides the tool for computing the minimum and maximum path-activated constraints. Function *Time* represents the number of cycles to traverse $A^*_f(l_1)$ to $A^*_f(l_m)$, when passing through $A^*_f(l_2),\ldots,A^*_f(l_{m-1})$. Note that when constraining the decision variables with respect to a path-activated constraint, all the actions that can be scheduled in the path will be constrained. We denote by $a_1, \ldots, a_o$ the set of actions that are executed by the CFE while executing the thread $[l_1,\ldots,l_m]$, and we denote by $x_{1,j1}, \ldots, x_{o,jo}$ their respective decision variables, where $j_1, \ldots, j_o$ range over the set of possible cycles where action a_i can be scheduled. When computing the constraint for the minimum and maximum thread

execution time, we must exclude the assignments to the decision variables which invalidates the limit on execution time for the thread. More formally,

Corollary 5.1:

$$\mathbf{min}(n,l_1 \ldots l_m) = \sum_{j1} \sum_{j2} \ldots \sum_{jo} \wedge_i e(x_{i,ji}) \, [\mathrm{Time}(A^*_f(l_1),\ldots,A^*_f(l_m)) \geq n] = 1,$$

where actions a_1, ..., a_o are the actions whose decision variables x_{1j1}, ..., $x_{o,jo}$ become constrained when traversing l_1, ..., l_m for more than n-1 cycles.

Corollary 5.2:

$$\mathbf{max}(n,l_1 \ldots l_m) = \sum_{j1} \sum_{j2} \ldots \sum_{jo} \wedge_i e(x_{i,ji}) \, [\mathrm{Time}(A^*_f(l_1),\ldots,A^*_f(l_m)) \leq n] = 1,$$

where actions a_1, ..., a_o are the actions whose decision variables x_{1j1}, ..., $x_{o,jo}$ become constrained when traversing l_1, ..., l_m for less than n+1 cycles.

Figure 13 presents the algorithm for minimum path-activated constraint and Figure 14 presents the algorithm for maximum path-activated constraint. The algorithm for exact delay is not presented, since it can be easily derived from both of these algorithms, and it corresponds to the intersection of the constraints obtained for minimum and maximum path constraints.

In Figures 13 and 14, we traverse the finite-state machine presented in Figure 12 while traversing the transition relation T. For each state i of the finite-state machine of Figure 12, we keep the current transitions of the CFFSM leading to i in variable T_i. Since the CFFSM may make multiple transitions at the same time when $A^*_f(l_i)$ and $A^*_f(l_{i+1})$ are satisfied simultaneously, after computing a new T_i (represented by NT_i), we must propagate it as far as possible, or until this new transition can not be propagated any longer in the machine of Figure 12.

In the case of a minimum path-activated constraint, we compute the relations among decision variables when the machine is traversed in less than $n+1$ cycles, since these assignments will be invalid. For the case of a maximum path-activated constraint, we consider that only these assignments will be valid assignments. In that way, each computation of a path-activated constraint can be computed in a finite number of traversals ($O(m^2 n)$). The universal quantification of the Boolean variables in B guarantees that the constraints on the decision variables will satisfy all assignments to the decision variables that can be constrained from each action of l_i. Finally, because complementing a set of states can generate a state which is not valid, every time we complement a set of states we intersect it with the variable *Valid*, which represents the set of reachable states of the CFFSM.