ANALYSIS AND SYNTHESIS OF CONCURRENT DIGITAL SYSTEMS USING CONTROL-FLOW EXPRESSIONS

A DISSERTATION

SUBMITTED TO THE DEPARTMENT OF ELECTRICAL ENGINEERING

AND THE COMMITTEE ON GRADUATE STUDIES

OF STANFORD UNIVERSITY

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

Claudionor José Nunes Coelho Junior

February, 1996

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.

Giovanni De Micheli(Principal Adviser)

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.

David Dill(Associate Adviser)

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.

Bruce Wooley

Approved for the University Committee on Graduate Studies:

Abstract

We present in this thesis 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 generate controllers 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 will be 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.

In order to generate the control-units for the design, we present two scheduling procedures. The first procedure, called static scheduling, attempts to find fixed schedules for operations satisfying system-level constraints. The second procedure, called

dynamic scheduling, attempts to synchronize concurrent parts of a circuit description by dynamically selecting schedules according to a global view of the system.

Dedication

To Carla and Jean-Luc,

Acknowledgments

I have many people to thank for this dissertation. First, in order to fulfill one of Prof. Giovanni De Micheli's last requests as an advisor, I shall be brief. So, instead of saying all good things one usually says about advisors, I will just say that I consider myself very fortunate to have him as my advisor and as a friend.

I would like to thank Prof. David Dill for all the help and support as a co-advisor. I am also very thankful to him for allowing me to participate in the verification group meetings and discussions, where I learned a lot. Prof. Bruce Wooley has been generous with his time by reading this dissertation. I would like to thank also Prof. Teresa Meng and Prof. Gordon Kino for serving on my oral defense committee.

I have many thanks to the members of the CAD and verification groups at Stanford. Many good ideas and discussions followed the interaction with them, especially with Luca Benini, Dave Filo, Rajesh Gupta, David Ku, Vincent Mooney, Polly Siegel, Han Yang and Jerry Yang. I would like to thank Toshiyuki Sakamoto, who has been working on the implementation of the *Parnassus* system. I have no words to thank Lilian Betters for making my life at Stanford so much easier by dealing with the university bureaucracy. Charlie Orgish and Thoi Nguyen have been very helpful in keeping the machines working. I have also to thank the people with whom I shared my office in the trailers and at CIS (Dave Ofelt, Jeff Kuskin, John Heinlein, and Wingra Fang) for making a friendly office environment.

Many friends I made at Stanford came from outside the research environment.

Thanks to all of them, specially to the lunch group, Alexandre Santoro and João Comba, for making my lunch hours much more relaxing and enjoyable. Thanks also to the several friends I have met here and who helped me throughout these years: Antônio Todesco, Luis Portela, Ciro Noronha, Tyiomari, Angela Comba, Felipe Guardiano, Alvaro Hernandez, Beto Cimini, Leda Beck, Ingrid and Luiz Franca. Thanks also to Gilberto Mayor, Diógenes Silva, Berthier Ribeiro and Rodolfo Resende for the friendship prior to my Stanford days.

I would like to thank my parents for believing in me and for all the support, guidance and encouragement. Thanks also to the support given by Maria, Florinda and by my in-laws Marcio, Domingos, Sirene and Carlos Vinicio.

Finally, I reserve a special gratitude to my beloved wife, Carla Nacif Coelho, for her support, love, patience and encouragement during our stay at Stanford, and for all the good moments we had here. Thanks to Jean-Luc, for bringing me the joy of being a parent. To them I dedicate this thesis.

This research was sponsored by the scholarship 200212/90.7 provided CNPq/Brazil, by a fellowship from Fujitsu Laboratories of America, and by ARPA, under grant No. DABT 63-95-C-0049.

Contents

List of Tables

List of Figures

Chapter 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 [Mic94].

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 [VRB+93]. 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 [KOH+94] and an ethernet coprocessor [HLS], 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 [MPC90], 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 [KM92]. This technique is called *single process synthesis* in [WTHM92], 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 dissertation 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.

- We present a technique for scheduling operations subject to complex interface constraints and synchronizations.

- We present a technique for synchronizing the concurrent parts of the design by dynamically scheduling operations or blocks of operations.

1.1 Overview of System-Level Synthesis

System-level design contains sub-components showing sequential, alternative, concurrent, repetition and exception handling behaviors [GVNG94]. We assume that the design is originally specified by a description language supporting these behaviors, such as VHDL [LSU89], Verilog HDL [TM91], HardwareC [KD90], StateCharts [DH86], Esterel [BS91].

Synthesis of system-level designs differs from standard high-level synthesis [DKMT90, CBH$^+$91, WTHM92, KLMM95] 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 [Gup93]. 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.

In system-level synthesis, partitioning involves the selection of clusters of operations and models that should be synthesized together [DGL92, TWL95] and clusters that should be synthesized separately. A good partition will be obtained by clustering parts of the specification that are tightly coupled and share the same critical resources of the design. This will allow the tools used at the later stages of the design to better optimize the cluster.

Another important task of system-level synthesis is scheduling. Scheduling denotes the assignment of operations over discrete time slots. Although high-level synthesis also considered scheduling as a synthesis task, we emphasize here the differences between scheduling in high-level synthesis and scheduling in system-level synthesis. In high-level synthesis, the main emphasis is put into the scheduling of operations within a basic block. Optimality of a design in high-level synthesis is usually given in terms of the optimality of the execution time in basic blocks or the cost of resources in basic blocks, such as the number of multipliers, adders or multiplexors. In system-level synthesis, on the other hand, we have to consider the interactions that cross basic block boundaries as well. When the system is partitioned in clusters some of the interactions of the system are converted into environment constraints, which should guide the tool in finding feasible and optimal implementation. Whenever these environmental constraints cross implementation paradigms (such as hardware and software), appropriate synchronization must be added as well. In addition to this, the optimality criteria in system-level synthesis shifts from basic blocks to whole parts of the design. For example, in a cache controller, the specification can be divided into a hit and miss case, both of which can share some parts of the specification. Since the hit case is going to be executed more often than the miss case, the primary optimization goal should be the minimization of the execution time of the hit path, and using the minimization of the execution time of the miss case as a secondary goal.

Synchronization and interface synthesis refers to the generation of protocol converters for some parts of the design. At the system-level of the specification, the user may have not committed to a protocol for the communication of the events across concurrent parts of the design, or the specification of the protocol may already exist in the form of libraries. In synchronization synthesis, the tool produces protocols for the communication among different parts of the specification, and generates converters between protocol libraries and the specification, according to the design constraints.

Finally, in datapath generation we obtain datapaths for the operations and their dependencies in the specification. During datapath generation, the tool selects components for an implementation, binds the components to operations, and binds the variables of the specification into registers.

Figure 1 gives an example of the tasks involved in the synthesis of system-level designs. From the specification of a concurrent system, a system-level tool first partitions the description and generates a set of crossing these partitions, then the tool schedules the operations over a discrete time according to the environmental constraints, synthesizes the synchronization skeletons and protocols for the different parts of the specification, and generates datapaths for the operations and variables.

1.2 Synthesis Tools used for System-Level Designs

Many systems implemented by Application Specific Integrated Circuits (ASICs) are control-dominated applications [Keu89]. In such applications, high-level synthesis techniques have been used previously to synthesize control-units for system-level designs.

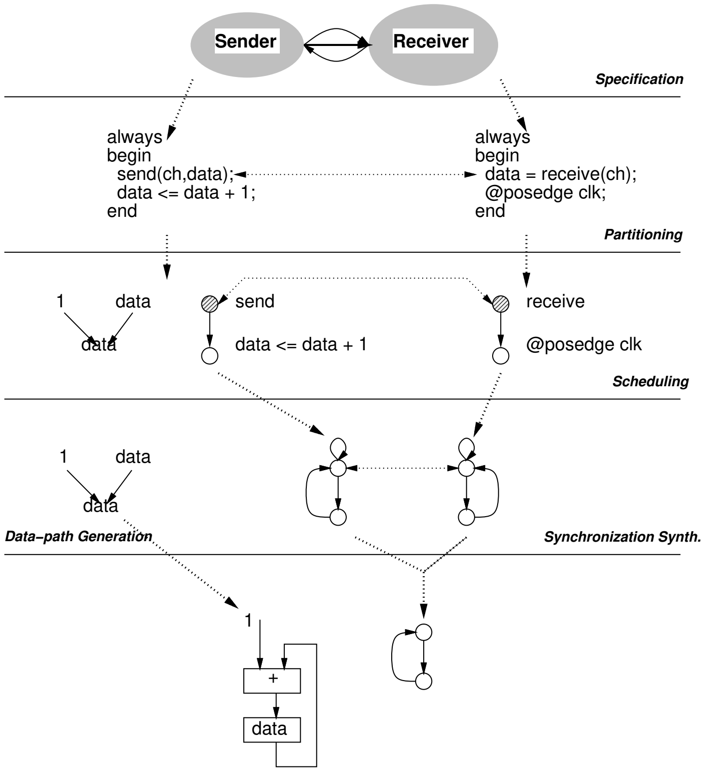


Figure 1: *System-level tasks*

The Olympus Synthesis System [DKMT90] 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 [CBH+91] 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 [Cam91]. 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 [WTHM92] (PUBSS) and the Synopsys Behavioral Compiler [KLMM95] 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 [Sea94] 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

[1] The muse of comedy

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, and flexible objective functions, which will help us to better cast our design goal.

In the next section, we will present some design problems that will help us to better understand the issues in system-level synthesis.

1.3 Issues in System-Level Synthesis

This section presents examples of designs that either cannot be synthesized or are synthesized sub-optimally by typical high-level synthesis tools. We show intuitively that valid and optimal implementations can be obtained only if synchronization, dynamic scheduling and scheduling with complex timing and resource constraints are considered during the design space exploration.

One of the major problems of using current synthesis tools to implement system-level designs is that the tool must consider how the environment affects the model being synthesized. 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 obtain optimal results. With *Thalia*, the user can specify complex environment constraints and flexible cost functions.

In the next three examples, we motivate the reader about the need for tools that can handle concurrent and communicating systems.

1.3.1 Synchronization Synthesis

In this example, we show how we can synchronize multiple processes sharing the same critical resource. We will see that this synchronization can be synthesized only if we consider the degrees of freedom among the different processes that share the critical resource. We are going to see that the model being synthesized will have to dynamically reconfigure itself in order to allow other models to use the same resource at different times. In this example, in order to obtain a feasible solution, we have to specify a constraint that spans across concurrent models, i.e., the critical resource should not be used by more than one model at a time.

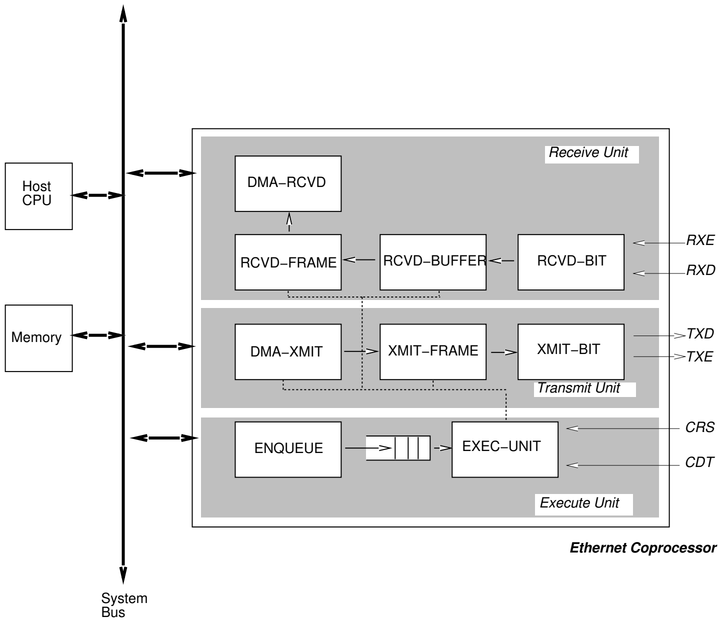


Figure 2: *Ethernet controller block diagram*

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, with three processes accessing the bus: *DMAxmit* , *DMArcvd* , and *enqueue*. The problem we want to solve is the synthesis of the 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 non-deterministic 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.

Related Work in Synchronization Synthesis

The problem of synchronizing critical resources across concurrent models has been solved for the simplified assumption that the models are dataflows executing at the same rate [HP92]. Note that in the problem described here, however, we do not know when each bus access will take place, since we may have arbitrary control-flow specifications that will make the bus accesses to be dependent on the environment and to execute at different rates. Thus, the approach described in [HP92] cannot be used for the bus accesses of the ethernet coprocessor described here.

Filo et al. [FKJM93] addressed the problem 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 subsume the solutions found by the two approaches previously discussed. Also, our formalism achieves 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 [FKJM93].

In [CE81], 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 [Wol82], 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 chapter, which will consider true concurrency. As a result, the synchronizers generated by these procedures will be subject to much stricter constraints than they will experience.

Zhu et al. [ZJ94, ZJ93a] 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 [Zhu92] 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 [ZJ94], 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.

Synchronization Synthesis for the Ethernet Coprocessor

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. Figure 3 presents the behaviors we assume for these descriptions in this paper, in a pseudo-Verilog 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. The signal `transmission ready` is assumed to be set by the environment surrounding the three processes, and `free bus` represents the waiting period for process *enqueue*.

```
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
```

Figure 3: *Abstracted behaviors for DMArcvd, DMAxmit and enqueue*

The processes shown in the figure 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.

We assume that processes *DMArcvd* and *DMAxmit* have already been synthesized, and their cycle-based behaviors are presented in the figure. We are interested in obtaining a control-unit for process *enqueue* such that it will not have conflicting bus accesses with neither *DMArcvd* nor *DMAxmit*. Note that in order to synthesize the waiting period for *enqueue* we must know when the other process will access the bus. Therefore, *enqueue* must have a global view of the bus accesses of the other processes in order to decide when it can access the bus.

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
   @ ( posedge clock);
   if (transmission ready)
   begin
      @ ( posedge clock);
   end
   else begin
      read bus;
   end
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*.

1.3.2 Scheduling under Complex Interface Constraints

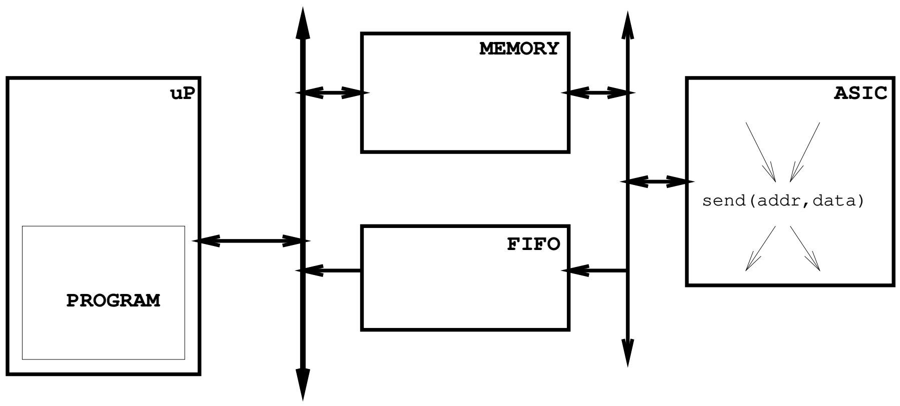


Figure 4: *System architecture*

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 [NG95, KM92, NT86, Bor88]. 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 Figure 4. 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 [GJM92, GJM94]. 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

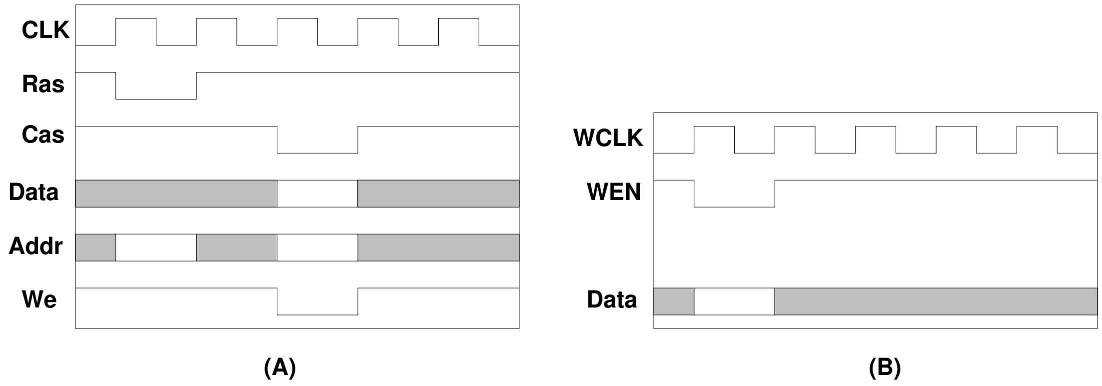


Figure 5: *Writing cycles for synchronous DRAM (a) and for synchronous FIFO (b)*

given in Figure 5. 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 [NT86], Ku [KM92] and Borriello [Bor88]. They used min/max scheduling constraints to annotate the design specification. The use of these constraints, however, is limited 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:

synchronize with "send" operation
if (s)
  delay for "send" is 3 cycles
else
  delay for "send" is 1 cycle

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.

1.4 Objectives and Contributions

In this thesis, we present a formal model to analyze system-level designs targeted to control-flow intensive applications, and a methodology to synthesize the control-units for the concurrent parts of the design. Because many applications found in Application Specific Integrated Circuits are control-dominated applications [Keu89], we will address the following issues regarding control-flow dominated system-level designs.

- *Modeling.* We will present a model to represent the control-flow of concurrent systems that includes most of the control-flow constructs present in specification languages, such as sequential, parallel and alternative compositions and loops. In addition to that, our model will support exception handling mechanisms which are present in languages such as Verilog HDL, Esterel and StateCharts. We will also allow systems to be specified with programming languages, such as C. Finally, we will include in our model some of the variables of the specifications, since in some cases these variables can give a better understanding of the control-flow behavior in such systems.

- *Constraints.* We will show how we can incorporate complex constraints of the design. The constraints of the design will not be limited to the constraints usually specified in high-level synthesis tools, but we will also allow the model's environment to be specified and to synchronize with the model being synthesized.

- *Analysis.* We will present techniques for analysis of the specification and its environment in a finite-state machine representation, and we will show how we can efficiently represent this finite-state machine. In this finite-state machine, we will be able to detect when no control-unit can be obtained for a specification when the specification is composed with its environment.

- *Synthesis.* We will present two techniques for solving the scheduling and synchronization synthesis problems. In the first technique, we will schedule operations statically over time to satisfy complex interface constraints. In the second technique, we will dynamically schedule the interacting parts of the specification in order to synchronize them. These schedules and the specification can be used to obtain a control-unit for the circuit description that optimizes a design goal, while satisfying the environment constraints.

1.5 Thesis Outline

The outline of this thesis (which is also the outline of the tool we developed) can be seen in Figure 6.

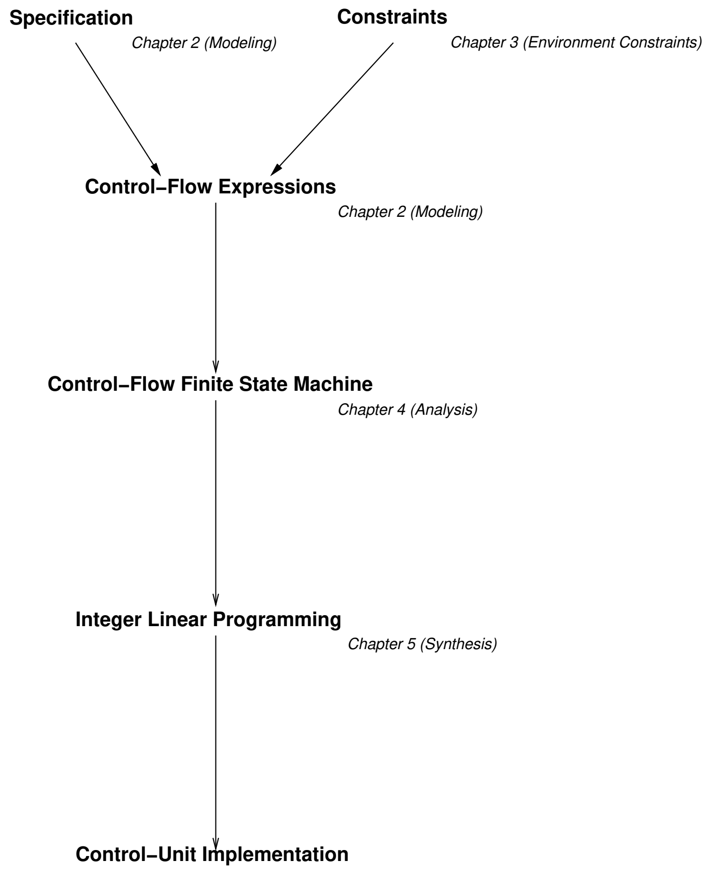


Figure 6: *Thesis outline*

After the introduction and motivation described in this chapter, Chapter 2 describes our model for concurrent control-dominated systems, called *control-flow expressions*. There, we present the algebra of control-flow expressions, and we show how this algebra can be used to model the control-flow aspects of a specification. Then, we present extensions to the algebra of control-flow expressions that will allow us to consider more realistic system-level designs, by incorporating some variables into the control-flow model, and by allowing the specification to contain exception handling mechanisms.

Since our model assumes that the system will be interacting with other models and with the environment, in Chapter 3 we will present techniques to incorporate design constraints into control-flow expressions. These constraints will include timing, resource and synchronization constraints.

Chapter 4 presents a method to analyze the system consisting of control-flow expressions by converting the system into a specification automaton that contains all degrees of freedom of the system being synthesized. We will also show how to represent this specification automaton in terms of a transition relation, and how to efficiently encode the different constructs of the control-flow expressions into the transition relation.

Chapter 5 describes two synthesis methods for scheduling operations and synchronizing parts of the description. Both of these algorithms are implemented as restrictions on the behavior of the specification automaton obtained in the previous chapter.

Chapter 6 presents some design examples and how they could be solved using the formulation presented in this thesis. Finally, in Chapter 7, we will present some concluding remarks and some ideas for future research.

Chapter 2

Modeling of Concurrent Synchronous Systems

We will be focusing in this chapter 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, an optimal controller can be obtained only if we understand the underlying behavior of the system to be synthesized, and its relation to the environment.

We model the system in terms of control-flow and dataflow for each concurrent component. We will first attempt to restrict the control-flow to the control-flow constructs of conventional structured languages, and we will restrict the dataflow model to the variables, and their corresponding operations. This abstraction model is presented in Section 2.1.

In Section 2.2, we present the algebra of *control-flow expressions*, which is an algebraic model for representing the control-flow of system-level designs, while abstracting away the dataflow details. In Section 2.3, the axioms of control-flow expressions will be introduced. These axioms form the basis for the analysis technique we will develop in the next chapter.

In order to better analyze the control-flow of system-level designs, it will be shown in Section 2.4 that variables and operations may play a fundamental role in defining the control-flow behavior, and we will show which parts of the dataflow must be considered during analysis and synthesis of control-flow dominated specifications. In addition to that, in order to capture basic blocks of traditional programming languages and hardware description languages, we will introduce blocks in control-flow expressions. Finally, exception handling mechanisms will also be added to control-flow expressions in order to capture the rupture of structured control-flow in the designs, usually due to the occurrence of exceptions that is common to most hardware description languages.

In Section 2.5, we compare extended control-flow expressions with existing formalisms that capture the control-flow of concurrent systems.

2.1 Abstraction Model

We consider in this thesis system-level designs that will be synthesized as synchronous digital 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 [DGL92, Mic94, ZJ93b]. In this model, variables, their operations and operands are placed in the dataflow, and the constructs determining the flow of control of the specification language are placed in the control-flow. I/O operations between a process and the process external environment will be placed in the dataflow. We formalize this model below.

Dataflow

We define a dataflow by its structure, without considering the meaning of the operations in the specification, similarly to the definitions of [ZJ93b]. Let $\mathcal{V} = \{v_1, \ldots, v_n\}$ be a set of variables, and let v be a generic element of $\mathcal{V}$. We assume that constant values are specified by variables whose names are represented by the constant value. Let $\mathcal{F}$ be a set of functions whose typical elements are f and f_i.

Definition 2.1 *An operation is defined as* $v \leftarrow f(v_1, \ldots, v_n)$*, i.e., variable* v *is assigned the result of function* f*, when the function's parameters are set to the variables* $v_1, \ldots, v_n$*.*

We call the set of operations $\mathcal{O}$. A dataflow can be defined as a set of operations and a partial order among them. It can be depicted as a directed acyclic graph in which vertices are operations and edges correspond to dependencies among the operations. Each edge is annotated with the variable that creates the dependency.

Control-Flow

In a hardware specification, as well as in a software program, the sequencing of the operations is determined by control-flow constructs, such as procedure calls, branching and iteration. In particular, descriptions can be made hierarchical by using procedure calls, which encapsulate portions of the behavior. Such a hierarchy may be abstracted as a directed acyclic graph, whose root corresponds to the overall system, whose internal vertices correspond to sequential, parallel, alternative and iterative compositions, and whose leaves are either operations, or groups of operations and dataflow models.

Different models [DGL92, Mic94, GVNG94] have been proposed to represent branching and iteration. In this thesis, we represent branching and iteration hierarchically, with their bodies being modeled as procedure calls, i.e. at a lower level in the representation hierarchy. Such calls are invoked conditionally according to the value of the branching or iterative clauses.

Note that in the control-flow and dataflow models defined above, the execution of the control-flow is data-dependent, and because the dataflow is conditionally executed, according to the control-flow, the dataflow is control-dependent. Since at this level of abstraction, the execution time for the operations is not known yet, in order to consider the communication between the dataflow and the control-flow we model the interface by instantaneous events. The control-flow generates output events to the dataflow that sensitize the execution of operations in the dataflow. The dataflow generates input events to the control-flow that trigger the different execution paths.

> **Example 2.1.1.** In Figure 7, 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 $a_1, \ldots, a_6$. 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 7 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. $\square$

We model a concurrent system by looking at the interface between the dataflow and the control-flow, i.e. at the events the control-flow generates and consumes. As a result, we need to abstract dataflow details from our model. We abstract the details

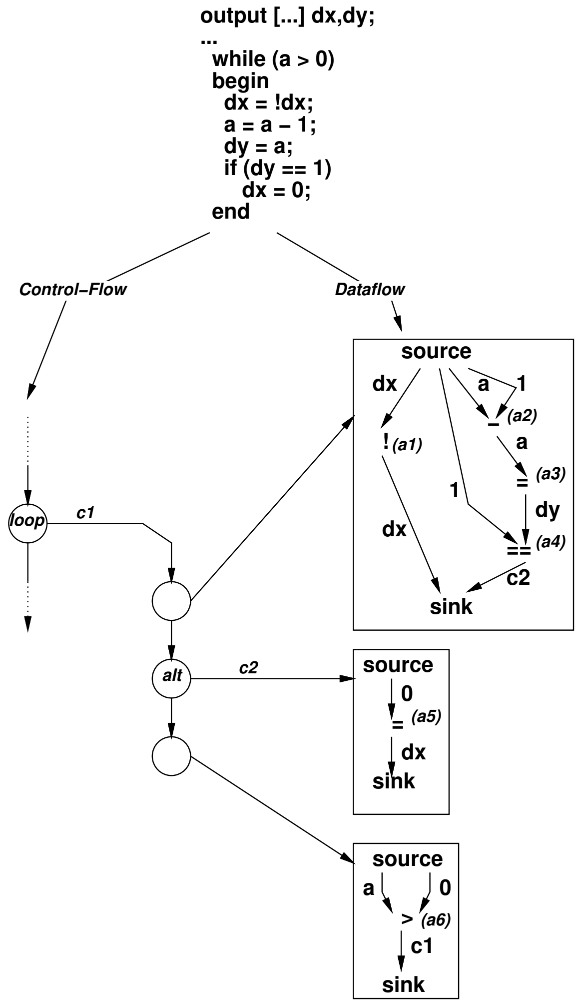


Figure 7: *Partitioning of specification into control-flow/dataflow*

from the dataflow by considering three mappings at the interface between the dataflow and control-flow: a timing mapping, a binding mapping and a synchronization mapping. The timing mapping associates an execution time with every computation or component. The binding mapping associates the possible functions of a computation with their possible implementations. Finally, the synchronization mapping specifies how the concurrent parts interact. The control-flow and the three mappings defined in this paragraph provide the means by which we can analyze the validity of the specification with respect to design constraints, as well as the means to generate possible implementations.

2.2 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 $\mathcal{A}$). 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 2.2.2. The C fragment presented below corresponds to a part of a differential equation solver found in [Mic94].

```
xl = x + dx;
ul = u - (3 * x * u * dx) - (3 * y * dx);
yl = 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 */
yl = 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 $\mathcal{C}$. 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 2.2 *A guard is a Boolean formula on the alphabet of conditionals. We will use $\mathcal{G}$ 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 2.2.3. In the specification `if (x ≥ y) x = y * z`, a conditional c abstracts the binary relational computation `x ≥ 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. □

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 $\mathcal{F}$ 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 HDLs which were captured by the control-flow model described earlier. Verilog HDL, for example, supports sequential composition, alternative composition, loops, parallelism 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 CFEs will also be guarded.

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

Definition 2.3 *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.*

We can now define the control-flow expressions recursively.

Definition 2.4 *Control-flow expressions are:*

- *Actions $a \in \mathcal{A}$.*
- *Processes $p \in \mathcal{P}$.*
- *δ and ϵ.*

- *If* $p_1, \ldots, p_n$ *are control-flow expressions, and* $c_1, \ldots, c_n$ *are guards, then the following expressions are control-flow expressions.*
  - *The sequential composition, represented by* $p_1 \cdot \ldots \cdot p_n$
  - *The parallel composition, represented by* $p_1 \| \ldots \| p_n$
  - *The alternative composition, represented by* $c_1 : p_1 + \ldots + c_n : p_n$
  - *Iteration, represented by* $(c_1 : p_1)^*$
  - *Unconditional repetition, represented by* p_1^ω.

Nothing else is a control-flow expression.

Informally, we define the behavior of the compositional operators of CFEs as follows: the sequential composition of p_1, ..., 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, ..., p_n means that all p_i's begins execution at the same time for $i \in \{1, \ldots, n\}$. The alternative composition of p_1, ..., p_n guarded by c_1, ..., 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.

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

[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.

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

Although this assumption seems to constrain the representation model using CFEs, in practice this will not impose problems because CFEs captures exactly the control-flow constructs of structured languages such as Verilog HDL and VHDL. With respect to C, we use a subset of C that does not allow a function to be defined recursively in order to avoid the possibility of having a CFE with process variables for which no CFE without process variables can be obtained. Later in this chapter we will enrich control-flow expressions by allowing CFEs to break the conventional flow of control, as in the case of breaks, continues and returns of the C programming language, or as in the case of disables of the Verilog HDL.

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

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

Table 1: *Link between Verilog HDL constructs and control-flow expressions*

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.

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 ($p, q \in \mathcal{F}$) and c is a conditional ($c \in \mathcal{C}$). In this figure, we relate CFEs to the control-flow structure of Verilog HDL [TM91]. In this thesis, 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 to that, 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.

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 $(\underbrace{p \cdot \ldots \cdot p}_{n})$, 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 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 $c : p + \overline{c} : q$) and executing the first action of process p, the execution of this action would make $\overline{c}$ *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* [BS91].

Assumption 2.1 *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 $(c : \epsilon) \cdot p$. 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.*

2.3 Axioms of Control-Flow Expressions

The algebra of control-flow expressions inherits its formalism from a subset of process algebras [Bae90] that is suitable for describing synchronous reactive systems, called the *algebra of regular synchronous processes.* We further extend this algebra by specifying Boolean variables as guards of processes. We refer the reader to Appendix A for a definition of the algebra of synchronous processes similar to the one found in [Bae90]. The following proposition relates control-flow expressions to the algebras of synchronous processes.

Proposition 2.1 *CFEs are a subset of regular synchronous process algebras.*

In this section, we present the axioms for the algebra of control-flow expressions by extending the axioms definitions of process algebras to handle actions and conditionals. These axioms provide the theoretical background that will be used to build the finite-state machine representation for control-flow expressions in Section 4.1.

In Table 2, we present the axioms of control-flow expressions, where a and b are 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 an action in 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 upon start by propagating the guards towards 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 non-causal (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.

$$
\begin{array}{rcll}
c_1 : p + c_2 : q & = & c_2 : q + c_1 : p & (+ \textit{ is commutative}) \\
(c_1 : p + c_2 : q) + c_3 : r & = & c_1 : p + (c_2 : q + c_3 : r) & (+ \textit{ is associative}) \\
 & = & c_1 : p + c_2 : q + c_3 : r & \\
(c_1 : p + c_2 : q) \cdot r & = & c_1 : p \cdot r + c_2 : q \cdot r & (\cdot \textit{ distributes to the left with } +) \\
(p \cdot q) \cdot r & = & p \cdot (q \cdot r) & (\cdot \textit{ is associative}) \\
 & = & p \cdot q \cdot r & \\
c_1 : p + c_1 : p & = & c_1 : p & (+ \textit{ is idempotent}) \\
 & & & \\
1 : p & = & p & \\
0 : p & = & \delta & \\
c_1 : p + \delta & = & c_1 : p & (\delta \textit{ is the identity element for } +) \\
\delta \cdot p & = & \delta & (\delta \textit{ is the zero element for } \cdot) \\
 & & & \\
p \cdot \epsilon & = & p & (\epsilon \textit{ is the identity element for } \cdot) \\
\epsilon \cdot p & = & p & \\
 & & & \\
c_1 : c_2 : p & = & (c_1 \wedge c_2) : p & \\
a || b & = & (a \cup b) & \text{if } a \cup b \text{ synchronize} \\
a || b & = & \delta & \text{if } a \cup b \text{ does not synchronize} \\
a || b & = & b || a & \\
a || 0 & = & a & \\
a || \epsilon & = & a & \\
a \cdot p || b \cdot q & = & (a || b) \cdot (p || q) & \\
a \cdot p || b & = & (a || b) \cdot p & \\
(c_1 : p + c_2 : q) || r & = & c_1 : (p || r) + c_2 : (q || r) &
\end{array}
$$

Table 2: *Axioms of control-flow expressions*

The parallel composition assumes *synchronous execution semantics*, also known as maximal parallelism semantics. In these 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 $\{a_1, \ldots, a_n\}$, where each $a_i \in \mathcal{A}$, actions $a_1, \ldots, a_n$ 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 $a = \{a_1, \ldots, a_n\}$ and $b = \{b_1, \ldots, b_m\}$ are composed in parallel, the resulting multiset $\{a_1, \ldots, a_n, b_1, \ldots, b_m\}$ 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 CFEs, 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 chapter, we will give an informal definition that will allow the reader to understand its meaning.

Processes synchronize in control-flow expressions in two ways. The first way is by defining multisets of actions that always have to execute at the same time, or by defining multisets of actions that should never execute at the same time. The second method of synchronization is achieved by defining guards that generate a deadlock when conjoined.

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.

Example 2.3.4. We provide here an example of the representation of Verilog HDL constructs in control-flow expressions. The specification shown in Figure 8 consists of an algorithmic representation of a greatest common divisor. Its control-flow expression is represented by process p, where the labels on the right correspond to the actions being executed or the conditionals on alternative compositions.

$$p \quad = \quad [(r : 0)^* \cdot b \cdot (c_1 : (c_2 : (c_3 : c)^* \cdot d)^* \cdot e + \overline{c_1} : \epsilon)]^\omega$$

□

2.4 Extended Control-Flow Expressions

In the previous section, we presented the basic constructs for control-flow expressions capturing most of the control-flow constructs of structured languages. In this

```
module GCD(Xin, Yin, ready, result);
  input [7 : 0] Xin, Yin;
  input ready;
  output [7 : 0] result;
  reg [7 : 0] result, x, y;

  always
  begin
    wait (ready)                 // conditional r
      {x, y} = {Xin, Yin};       // action b
    if (x! = 0 && y! = 0)        // conditional c1
    begin
      while (y! = 0)             // conditional c2
      begin
        while (x >= y)           // conditional c3
          x = x − y ;            // action c
        {x, y} = {y, x} ;        // action d
      end
      result = x ;               // action e
    end
  end
endmodule
```

Figure 8: *Greatest-common divisor example*

section we revise the control-flow expression model, and we extend it to incorporate exception handling mechanisms encountered in hardware description languages. Extended control-flow expressions (ECFEs) will be formally defined in Section 2.4.4, after having presented their motivation and usage.

One interesting observation about the control-flow expression model we developed in the previous sections is that it captures the control-flow of series-parallel [2] representation of systems. In imperative programming languages, however, blocks of operations and their dependencies usually cannot be represented efficiently using this series-parallel representation, and as a result, in order to efficiently consider them in our model we will add basic blocks to CFEs.

[2]We denote by a series-parallel system the representation of a system in terms of mixture of sequential and parallel compositions

Finally, as shown in Section 2.1, we modeled the communication between the control-flow and dataflow through events. Sometimes, this communication can be reduced if some of the variables mapped into the dataflow are considered during the control-flow analysis, such as variables representing states of program-state machines or counters. We will present in this section the general theory under which such deviations from the traditional approach to control/dataflow partitioning can be made.

As opposed to the axioms presented in the previous sections, the properties for the extended control-flow expressions will be presented only in Chapter 4, since the system's properties will be better understood at the finite-state machine level, which is defined in Chapter 4.

2.4.1 Exception Handling

We assume the system-level design was specified by a structured language such as C, Verilog HDL or VHDL, with limited forms of disruption of the control-flow. In particular, all compositions defined for CFEs have single exit points. In this section, we enrich CFEs to allow systems to be described using multiple exit points from the control-flow. This extension will add to the efficiency in the representation of more complex control-flow structures, such as exception handling mechanisms that are present in reactive systems.

Examples of the disruption in the control-flow of structured languages can be found in both C and Verilog HDL. In C specifications, *returns*, *breaks* and *continues* are used to exit the procedure (as in the case of *return*) or the inner most loop (as in the case of *break* or *continue*). In Verilog HDL, multiple-exits from the control-flow corresponds to a limited form of the *disable* statement.

The exception handling mechanism added in this section is called *disable* (for its similarities with the *disable* statement in Verilog) that allows the control-flow to be disrupted at any point in the hierarchy. As it was shown in the control-flow model presented earlier, the control-flow of a specification can be viewed as a hierarchical composition of other control-flows. Similarly, the corresponding CFE can be seen as a hierarchical composition of other CFEs. In particular, the *uplink* of a control-flow expression represents a snapshot of the hierarchy with respect to a control-flow expression.

Definition 2.5 *Let p and q be control-flow expressions. An $n -$ uplink relation between p and q can be defined recursively as follows.*

- *Control-flow expression q is a 0-uplink of p if $q = p$.*
- *If q_i is a n-uplink of p, then q is a $n+1$-uplink of p in the following CFEs.*
  - $q = q_0 \cdot \ldots \cdot q_i \cdot \ldots \cdot q_n$;
  - $q = c_0 : q_0 + \ldots + c_i : q_i + \ldots + c_n : q_n$;
  - $q = q_0 || \ldots || q_i || \ldots || q_n$;
  - $q = q_i^\omega$;
  - $q = (c_i : q_i)^*$.

The definition above defines q as an *uplink* of a CFE p by counting how many compositions of control-flow expressions are necessary to obtain CFE p containing q. This definition is better explained by the example below.

> **Example 2.4.5.** The control-flow expression for the GCD algorithm of Example 2.3.4 is presented below, and its directed acyclic graph representation is presented in Figure 9, according to the composition rules for CFEs. In this graph, nodes represent control-flow expression composition rules, with actions and ϵ being the terminal vertices. The edge (p, q) represents the relation *the control-flow p can be obtained by a composition of q.*
>
> $$p = [(r : 0)^* \cdot b \cdot (c_1 : (c_2 : (c_3 : c)^* \cdot d)^* \cdot e + \overline{c_1} : \epsilon)]^\omega$$
>
> The following uplinks are defined for action c in CFE p.

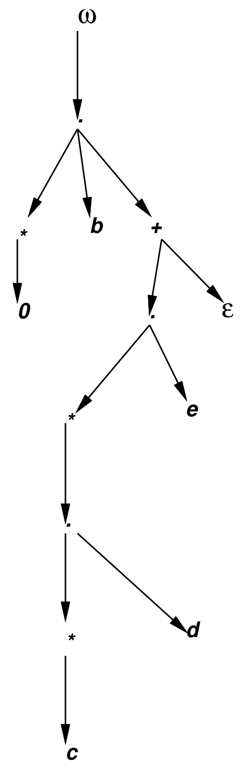


Figure 9: *Hierarchical View of a CFE*

- c is a *0-uplink* of c in p.
- $(c_3 : c)^*$ is a *1-uplink* of c in p.
- $(c_3 : c)^* \cdot d$ is a *2-uplink* of c in p.
- $(c_2 : (c_3 : c)^* \cdot d)^*$ is a *3-uplink* of c in p.
- $(c_2 : (c_3 : c)^* \cdot d)^* \cdot e$ is a *4-uplink* of c in p.
- $(c_1 : (c_2 : (c_3 : c)^* \cdot d)^* \cdot e + \overline{c_1} : \epsilon)$ is a *5-uplink* of c in p.
- $(r : 0)^* \cdot b \cdot (c_1 : (c_2 : (c_3 : c)^* \cdot d)^* \cdot e + \overline{c_1} : \epsilon)$ is a *6-uplink* of c in p.
- p is a *7-uplink* of c in p.

□

The *uplink* of a control-flow expression is used to determine where in the hierarchy the control-flow should continue the execution after executing the *disable* operation.

Definition 2.6 *$\zeta(n, p)$ is an extended control-flow expression that takes zero time and returns the control-flow to the ECFE following the n-uplink of $\zeta(n, p)$ in p.*

This definition can be seen as the execution of the extended control-flow expression following the *n-uplink* of ζ, when the ECFE represented by the *n-uplink* of ζ is replaced by an ϵ. The definition of ζ above introduces a technicality into the axioms of control-flow expressions. Recall that $((p_1 \cdot p_2) \cdot p_3) \equiv (p_1 \cdot p_2 \cdot p_3)$. In order for this axiom to be valid when we consider ECFE ζ, we have to compute the uplink with respect to the original specification p, and not with respect an ECFE p' equivalent to p. Thus, if p is a *n-uplink* of $\zeta(m, p)$ in p, and even though $p \equiv p'$, we still keep the expression $\zeta(m, p)$ in p'.

We will refer to ECFE ζ as the disable construct in this thesis. The definition above for the disable construct allows us to represent C's breaks, continues and returns, as seen in the following propositions.

Proposition 2.2 *A break command in C can be converted into a disable operation in extended control-flow expressions, where n is the least uplink enclosing a loop, an infinite computation or an alternative composition corresponding to a switch command in the extended control-flow expression representing the control-flow of the C program.*

Proposition 2.3 *A return command in C can be converted into a disable operation in extended control-flow expressions, where n is the largest uplink in the extended control-flow expression representing the extended control-flow of the C program.*

Proposition 2.4 *A continue command in C can be converted into a disable operation in extended control-flow expressions in the following way. Break the block of the least n-uplink containing a loop or infinite computation into two blocks located at uplinks n and $n+1$, respectively.*

Example 2.4.6. In Figure 10, we see an example for the definition of the disable constructs in extended control-flow expressions in order to represent the rupture in the control-flow of C programs.

The ECFE corresponding to the C fragment of Figure 10 is presented below, where actions a, b and c represent operations `i=0`, `a[i]++` and `i++`, respectively, and conditionals c_1, c_2, c_3 and c_4 denote the result of operations $\texttt{i} < \texttt{N}$, $\texttt{a[i]} > \texttt{0}$, $\texttt{i} > \texttt{10}$, and $\texttt{k} < \texttt{i}$, respectively.

$$p = a \cdot (c_1 : ((c_2 : \zeta(4,p) + \overline{c_2} : \epsilon) \cdot (c_3 : \zeta(2,p) + \overline{c_3} : \epsilon) \cdot (c_4 : \zeta(5,p) + \overline{c_4} : \epsilon) \cdot b) \cdot c)^*$$

□

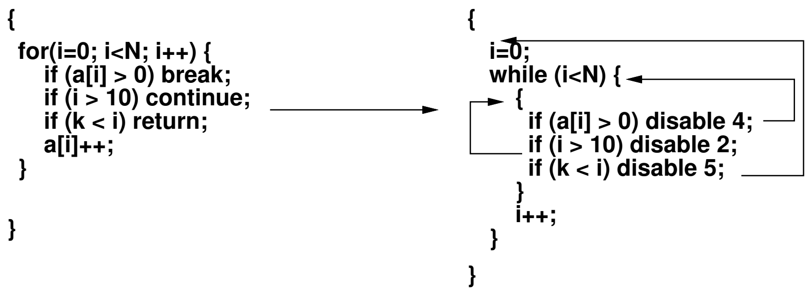


Figure 10: *Conversion between C constructs and ECFE disable constructs*

With this enhancement of control-flow expressions we are able to represent all control-flow constructs of the C programming language, with the exception of *goto*, which completely unstructures the control-flow of a C program.

In Verilog HDL, we are able to represent a limited form of the Verilog disable construct, when restricting the disable statement of Verilog to the specification hierarchy. This disable construct allow us to represent most of the interesting cases for exception handling using disable, with the notable exception of a disable block terminating the execution of a block executing concurrently with the block containing the disable operation. However, we can always rewrite the specification in such a way that both blocks are disabled during the termination of the execution.

Example 2.4.7. The Verilog code of Figure 11 represents a controller that puts a frame in block of data, as in the case of a communications controller.

```
fork : BLK_0
  begin : BLK_00
    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 BLK_0;
  end
  begin : BLK_01
    wait (posedge CCT);
    disable BLK_0;
  end
join
```

Figure 11: *Exception handling in Verilog HDL*

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 *BLK_00*, indicating that the transmission has been interrupted. Conditional len corresponds to the result of the comparison `length` $>$ `0`.

The ECFE for this Verilog is presented below, where p_{00} is the ECFE for block *BLK_00* and p_{01} is the ECFE for block *BLK_01*.

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

□

It is worth noting that this type of exception handling mechanism fits perfectly well into the framework of other languages for reactive systems, such as Esterel [BS91] or StateCharts [DH86].

2.4.2 Basic Blocks

The definition of control-flow expressions presented in Section 2.2 presents some deficiencies for representing dataflow graphs, i.e., graphs that can not be easily represented by series-parallel compositions of control-flow expressions. One of the possible ways of representing dataflow graphs is the enumeration of all paths of the graph, making these paths sequential compositions of CFEs, and composing them in parallel, with the proper synchronization. This approach, however, makes the representation of dataflow blocks inefficient, since it will increase the complexity of the control-flow expression representation. We propose in this section the addition of blocks to the algebra of control-flow expressions in order to efficiently encapsulate dataflow graphs.

Definition 2.7 *Let $\mathcal{A}$ be a set of actions, and let the relation $\rightarrow$: $\mathcal{A} \times \mathcal{Z} \times \mathcal{A}$ represent precedence constraint between two actions, where $\mathcal{Z}$ is the set of integer numbers. Then, $a_1 \stackrel{n}{\rightarrow} a_2$ 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 2.8 *Let r_i be a precedence constraint. A basic block is an extended control-flow expression represented by the set of precedence constraints $\{r_1, r_2, \ldots, r_m\}$.*

Note that this basic block can always be represented by the enumeration of all paths, thus, the basic block definition in extended control-flow expressions only adds a compact and efficient representation for dataflow representations. We will denote a generic basic block by $\{\mathcal{R}\}$.

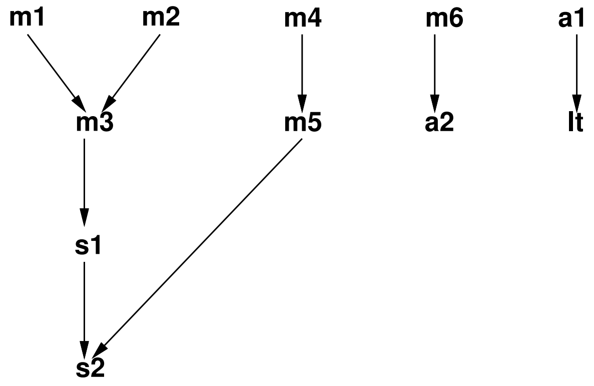


Figure 12: *Dataflow for Differential Equation Fragment*

Example 2.4.8. The dataflow graph of Example 2.2.2 is presented in Figure 12. This dataflow can be represented by the extended 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:

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

□

2.4.3 Register Variables

In order to define the last extension to control-flow expressions, let us consider Figure 13. 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.

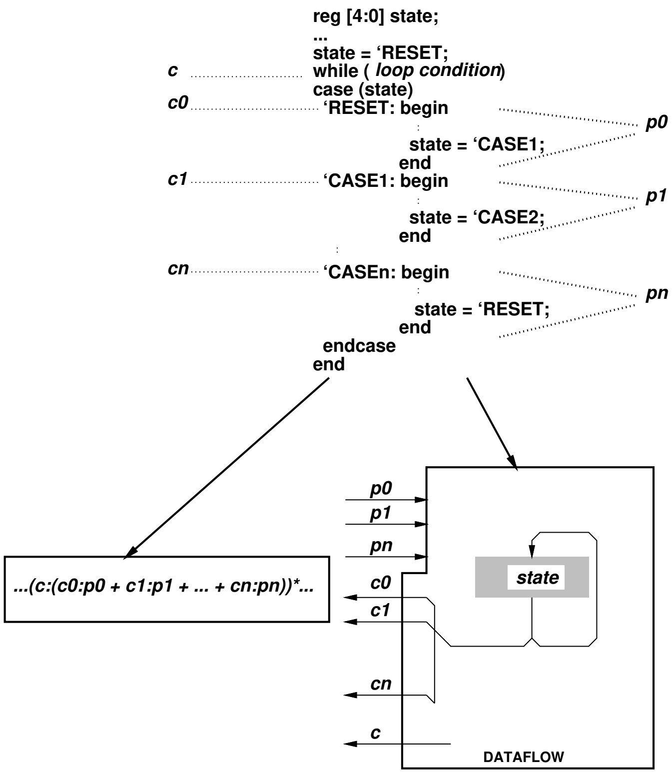


Figure 13: *Program-State Machine Specification*

Control-flow/dataflow transformations have been regarded in the past as useful transformations [DDT83, Sta70, DGL92]. 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.

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

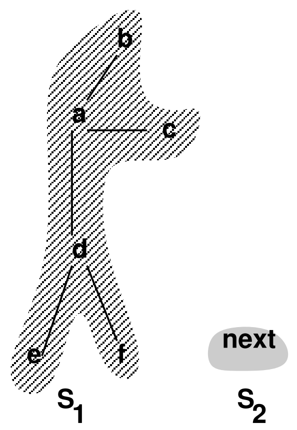
Definition 2.9 *A reduced dependency graph is the undirected graph* $G_r = (V_r, E_r)$, *where V_r is the set of non-constant variables, and an edge between two variables u and v exists if u depends on v or if v depends on u in at least one of the dataflows of $\mathcal{D}_f$.*

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 $S = \{S_1, \ldots, S_n\}$ 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.

```
if (c > 0) then
begin
   a = b + c;
   next = 3;
   d = e + f;
end
else
begin
   a = d + 2;
   next = 4;
end
```

(a)



(b)

Figure 14: *(a) Specification and (b) Reduced dependency graph*

Example 2.4.9. In Figure 14 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 ECFE 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 $\sigma = \{v, c_1, \ldots, c_m\}$ 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 2.1 *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 $c_j \leftarrow \rho(v_1, \ldots, v_n)$ 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 13 satisfies the conditions of Corollary 2.1. We present in Figure 15 (a) the different dataflows for the description of Figure 13, and in Figure 15 (b) the reduced dependency graph for these dataflows. Other variables that often occur in the specifications of control-dominated specifications are counters, for example.

The observations shown in this section leads to the following extension to control-flow expressions.

Definition 2.10 *Let σ be the connected component of V_r defined previously. Then $[v \leftarrow \gamma]$, $[v++]$ and $[v--]$ are extended control-flow expressions, and $[\rho(v, \gamma)]$ is a guard.*

Note 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 $\{0, \ldots, |v| - 1\}$, where $|v|$ is the number of possible values for the register.

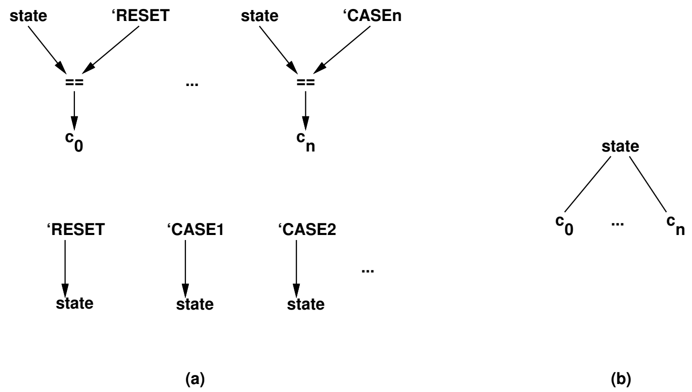


Figure 15: *(a) Dataflow graphs for program-state machine and (b) reduced dependency graph*

We could have extended the model presented above to incorporate more of the dataflow of the specification into the control-flow. However, introducing more variables into the extended control-flow expressions could easily increase complexity in the internal representation of the control-flow.

2.4.4 Definition of Extended Control-Flow Expressions

We can now define extended control-flow expressions, by putting together all the elements from control-flow expressions with the extensions proposed in this section.

The set of symbols is extended to include registers, i.e., $\Sigma = \mathcal{A} \cup \mathcal{C} \cup \mathcal{R} \cup \mathcal{F}$, where $\mathcal{A}$ is the set of actions, $\mathcal{C}$ is the set of conditionals, $\mathcal{R}$ is the set of register variables with a finite number of possible values and $\mathcal{F}$ is the set of process variables. The set of operations is defined as $\mathcal{O} = \{\cdot, +, :, *, \omega, \|, \zeta, [:=], [++], [--], \{\mathcal{R}\}\}$, where $\{\cdot, +, :, *, \omega, \|\}$ are the compositions of control-flow expressions defined in Section 2.2, ζ is the disable construct, $[:=], [++]$ and $[--]$ are functions defined on registers, and $\{\mathcal{R}\}$ are basic blocks. The formal definition is presented below.

Definition 2.11 *Let $(\Sigma, \mathcal{O}, \delta, \epsilon)$ be the algebra of extended control-flow expressions where:*

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

$\mathcal{O}$ 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 extended control-flow expressions are defined also as Boolean functions over conditionals, but including relational operations between registers and constants.

We can now define a more general form of control-flow expressions, which will be called in this thesis *extended control-flow expressions*, or ECFEs.

Definition 2.12 *Extended control-flow expressions are:*

- *Actions $a \in \mathcal{A}$.*
- *Process $p \in \mathcal{F}$.*
- *δ and ϵ.*
- *$\zeta(n, p)$, where n is a natural number and $p \in \mathcal{F}$.*
- *$\{r_1, r_2, \ldots, r_m\}$, where r_i is a precedence relation of the form $a_j \xrightarrow{n} a_k$, $\{a_j, a_k\} \subseteq \mathcal{A}$.*
- *$[v := constant], [v++], [v--]$, where v is a register*
- *If $p_1, \ldots, p_n$ are extended control-flow expressions, and $c_1, \ldots, c_n$ are guards, then the following expressions are extended control-flow expressions.*

– *The sequential composition, represented by* $p_1 \cdot \ldots \cdot p_n$

– *The parallel composition, represented by* $p_1 \| \ldots \| p_n$

– *The alternative composition, represented by* $c_1 : p_1 + \ldots + c_n : p_n$

– *Iteration, represented by* $(c_1 : p_1)^*$

– *Unconditional repetition, represented by* p_1^ω.

Nothing else is an extended control-flow expression.

Unless otherwise stated, we will refer to extended control-flow expressions by control-flow expressions in the remainder of this thesis.

2.5 Comparison of CFEs with Existing Formalisms

Control-flow expressions are very useful as a modeling and abstraction formalism for the control-flow of a concurrent system, since the translation from control/dataflow representation into CFEs is straightforward. In this section, we compare CFEs with other representative 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, CIRCAL and BFSMs.

- The algebra of **regular expressions** [Kle56, HU79] is used to represent strings accepted or 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 [Pai77, Las90]. In [TU82], regular expressions were used as a specification language for finite-state machine recognizers. 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 non-determinism [Mil84]. 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, a requirement for a system to react correctly to its environment.

CFEs also extends regular expressions by defining infinite behaviors, which could be achieved only by extending regular expressions to ω-regular expressions [Cho74], by breaking the normal flow of execution by means of the disable construct, and by allowing certain variables from the dataflow to be incorporated into the control-flow.

- **Path expressions** [Cam76] 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. Control-flow expressions extend path expressions by providing guards to alternative branches and loops, in the same way CFEs extended regular expressions. CFEs also breaks the flow of control with the disable construct and it allows certain variables from the dataflow to be incorporated into the control-flow.

- A **finite-state machine** [Mea55, Moo56, HU79] 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 [DH86] and SDL [Sar89]. 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. Furthermore, a finite-state machine model flattens the hierarchy present in the control-flow, thus preventing any control-optimization technique based in control-flow hierarchy [KM92] to be used.

- **Petri nets** [Pet81] 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 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 [Wan88]. 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.

Whereas Petri-nets only have a graphical representation, control-flow expressions views the system in an algebraic way, presented in this chapter, and in a graphical representation as a finite-state machine, which will be presented in Chapter 4. A dual view allows the analysis of a concurrent system to be performed at two levels of abstraction. In the algebraic view, we analyze the system by using its hierarchy (in terms of uplevels of a CFE) to abstract "uninteresting" portions of the computation. In the graphical representation, we perform reachability analysis that enables us to reason about the system as a whole. Together, these two procedures can be more efficient than just a graphical representation. Similar to CFEs, Garg et. al. [GG92] defined concurrent regular expressions as an algebraic model for Petri-nets. One of the key differences between CFEs and concurrent regular expressions is that the former will be able capture more succinctly the control-flow of systems described by hardware description languages and programming languages.

- **Process algebra** [Bae90] and **CCS** [Mil91] 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, non-deterministic choice, parallel composition and communication. Concurrency usually refers to interleaved concurrency, which is represented by non-deterministic 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 better capture the reactive nature of hardware systems, and as a result, control-flow expressions will fit better the model used for synthesis.

  When compared to the extensions to control-flow expressions presented in Section 2.4, we see that CFEs gives more flexibility to encapsulate the control-flow behavior present in systems that needs to react to its environment, sometimes in a non-structured way, such as with exception handling mechanisms or by representing state variables.

- **Timing expressions** [ZJ93a, ZJ94] is a model for *describing behaviors of sequential systems and specifying sequential constraints a sequential system has to satisfy* [ZJ93a]. 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 to that, control-flow expressions can be considered as a superset of timing expressions, since CFEs can represent systems containing hierarchical series-parallel specifications, whereas in timing expressions parallelism can occur only at the highest level.

- **CIRCAL** [Mil85, Mil94] is a model for concurrent systems in which alternative behavior and non-deterministic choice is represented separately. The representation of a concurrent system in CIRCAL consists of a set of labels executing concurrently — thus this model supports true concurrency — and the operations among these models, including sequential, alternative, non-deterministic choice, concurrency and communication, which is achieved by matching labels of a CIRCAL description.

  When representing a reactive system, CIRCAL does not distinguish the inputs from the outputs of the system. Thus, we can view a CIRCAL description as a high-level view of the interactions that occur in a system, without worrying about how these interactions take place. In CFEs, we represent the interactions of the inputs and the outputs in a system differently. Inputs of CFEs compose using regular Boolean laws, and are assumed to execute in zero time, whereas outputs of CFEs compose through synchronization, similar to the composition rules for CIRCAL.

- **BFSMs** [WTL91] are a generalization of finite-state machines with partial timing information on the relative execution time of the states. 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 [3] formalism and the lack of a formal model for constraint specification — which is restricted to scheduling constraints — prevents BFSMs from being used in more complex problems. As opposed to CFEs, 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 BFSMs, and thus, optimizations that would be best used at the expression level — such as hierarchical abstraction and rewriting — are not available to the synthesis process. Finally, a 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 control/dataflow abstractions — 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.

2.6 Summary

In this section we presented a model for representing the behavior of a control-dominated concurrent system. This model was called the algebra of control-flow expressions, and its axioms were presented along with a discussion of its limitations for representing the control-flow of structured programs.

[3] We will define formally *synchronization* in the next chapter.

We also provided extensions to the algebra of control-flow expressions so that we are able to represent exception handling, basic blocks and a limited form of register variables. Exception handling was introduced to overcome the limitation of a single-exit point from the structured compositions of CFEs, thus allowing us to represent the disruption of control. Basic blocks were defined to reduce the complexity for representing general dependency structures corresponding to basic blocks in programming languages. Register variables enriched control-flow expressions by enabling the representation of more diverse control-flow structure, such as in the case of a program-state machine. We showed how to systematically repartition the control-flow and dataflow portions in order to reduce the communication between control-flow and dataflow, thus creating a more accurate execution model for the control-flow when we abstract the dataflow.

These extensions allowed us to express efficiently most of the control-flow from structured programming languages and hardware description languages.

In the next chapter, we will focus on the analysis of a system modeled by control-flow expressions.

Chapter 3

Modeling the Environment

Characterizing a system for synthesis implies modeling the design and the environment surrounding it. We modeled the design in the previous chapter with CFEs. In this chapter, we will show how the environment can be represented using CFEs.

At the higher levels of abstraction, the specification is non-deterministic, because at this level we like to encapsulate multiple design choices of the system. This non-determinism is resolved during the synthesis of control-units by selecting a deterministic implementation that optimizes a design goal. As a result, we will have to uniquely identify the non-deterministic choices of the design when we model the system's non-determinism. This will be achieved by guarding CFEs with decision variables.

Since the specification of the environment surrounding a specification is a formidable task, we will be providing methods for abstracting the environment. In particular, we will be interested in abstractions that can be used for scheduling operations statically in basic blocks, and in abstractions that can be used to synchronize the concurrent parts of the design. Among these abstractions, we will show how to represent the environment's timing, resource usage and synchronization constraints.

3.1 Quantification of the Design Space

In this section, we will show how decision variables can be used to quantify the design space. Let us begin by defining what is a decision variable.

Definition 3.1 *A decision variable x 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 quantifying implementation choices for control-flow expressions. They can be assigned either constant values or Boolean functions over a set $\mathcal{D}$. At this point, we have to distinguish among these two types of assignments, and where they appear in control-flow expressions.

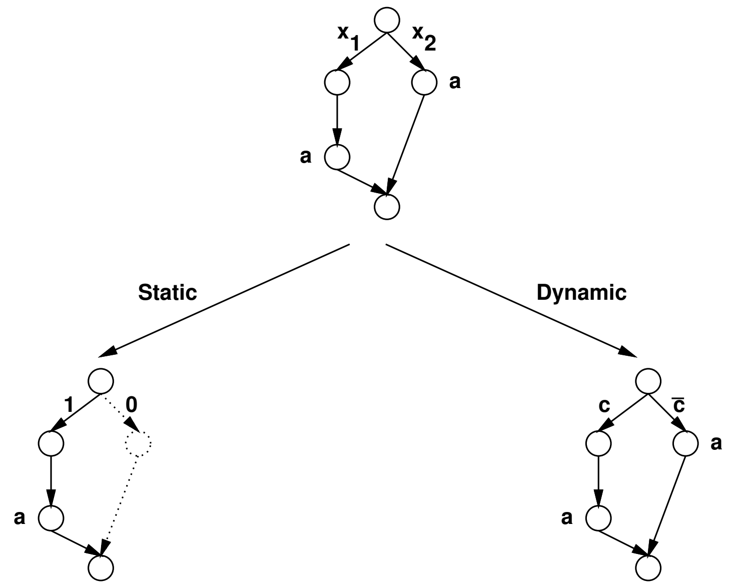


Figure 16: *Static and dynamic decision variables*

We call static decision variables the decision variables that are assigned constant values, and dynamic decision variables the decision variables that are assigned the Boolean functions over $\mathcal{D}$. Figure 16 shows the difference between static and dynamic decision variables. We assume that x_1 and x_2 are decision variables that will determine when some action a will be executed. If x_1 and x_2 are static decision variables, they will be assigned constant values such that only one of the possible executions for action a will be selected during synthesis. If x_1 and x_2 are considered to be dynamic decision variables, then both schedules are possible, and the selection will be made at execution time, based on some input c.

Although it may seem that it is always better to use dynamic decision variables, the algorithms for assigning values to them are not as efficient as the algorithms used for assigning values to static decision variables. As a result, we use both static and dynamic decision variables in CFEs.

Static decision variables are used to identify possible schedules for the operations inside basic blocks of CFEs. We will assume that basic blocks will encapsulate a set of operations (represented by the respective actions) that must be scheduled statically with respect to the beginning of the basic block.

Recall that a basic block is represented by a set of precedence constraints. According to these precedence constraints and to the maximum allowed time to execute a basic block, each operation of the basic block will have a set of possible times it can execute relative to the beginning of the basic block.

We annotate each possible execution time of an operation inside its basic block with a static decision variable, and this decision variable will uniquely identify when the operation will be executed. Suppose an operation o_i can be executed at times $\{j_1, \ldots, j_n\}$. Then, we create n decision variables for the operation o_i, namely $x_{ij_1}, \ldots, x_{ij_n}$. If x_{ij_1} is one, operation o_i is executed at time j_1.

We will also introduce a decision variable for each possible execution time of the basic block because a constraint satisfaction may require a basic block to execute beyond the minimum time necessary to execute its operations.

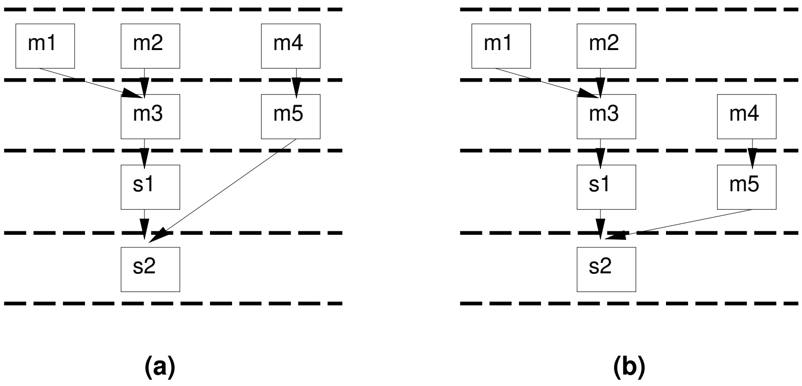


Figure 17: *Minimum (a) and maximum (b) execution times for the operations of the differential equation CDFG*

> **Example 3.1.1. Static Decision Variables:** Figure 17 presents the minimum (a) and maximum (b) execution times for a part of the control-data flow graph of the differential equation solver of Example 2.3.2. In this example, we restricted the maximum execution time for the basic block to four cycles.
>
> The basic block representing the system is given by $\{m_1 \rightarrow m_3, m_2 \rightarrow m_3, m_4 \rightarrow m_5, m_3 \rightarrow s_1, s_1 \rightarrow s_2, m_5 \rightarrow s_2\}$.
>
> In Figure 17, since action m_1 can only be executed in the first cycle, it only requires one decision variable, x_{m_1}. For action m_4, we represent the design space using two decision variables, $x_{m_4,1}$ and $x_{m_4,2}$. If $x_{m_4,1}$ is true, then operation m_4 is executed in the first cycle of the basic block. If $x_{m_4,2}$ is true, then operation m_4 is executed in the second cycle. The decision variables for the other actions can be obtained in a similar manner.
>
> In addition to the decision variables defined in the previous paragraph, we also define four decision variables corresponding to the four cycles of the basic block, namely decision variables y_1, y_2, y_3 and y_4. If y_i is *true*, for example, the basic block must execute in exactly i cycles, independent on the execution time necessary to execute the operations of the basic block. In this case, it should be clear that y_4 will be always *true*, and y_1, y_2 and y_3 will be *false*. If we had chosen a basic block executing in at most 5 cycles, then either y_4 or y_5 could be *true*. $\square$

We also introduce decision variables outside the scope of basic blocks. These decision variables will be called dynamic decision variables and they will be dynamically satisfiable. They will be assigned Boolean functions whose truth values change over time. In the following chapters, we will see that the assignment to these variables will consider the "state" of the system being synthesized, and this will enable us to synchronize the concurrent parts of the specification with respect to the design constraints.

Since static decision variables are only used to identify possible schedules inside basic blocks, and dynamic decision variables will be used elsewhere in the specification, we will refer to both as decision variables, and it will be clear from the context which type of decision variable we are referring to.

In control-flow expressions, decision variables can be used as guards of expressions, so we extend guards to allow decision variables and conditionals to be composed together.

Definition 3.2 *A guard is a Boolean formula over the set of decision variables and the set of conditionals.*

> **Example 3.1.2. Dynamic Decision Variable:** 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 is a decision variable. In this CFE, the possible schedules are quantified by the different assignments of the decision variable x over time.
>
> Possible assignments could be:
>
> $$\begin{array}{r} a \cdot b \\ a \cdot 0 \cdot b \\ a \cdot 0 \cdot 0 \cdot b \\ \vdots \\ a \cdot 0 \cdot \cdots \cdot 0 \cdot b \end{array}$$
>
> 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. □

Note the structure imposed by allowing dynamic decision variables in CFEs, and static decision variables in basic blocks. During synthesis, operations inside basic blocks are statically scheduled with respect to the beginning of the basic block, which can be dynamically synchronized with other parts of the design by guarding the basic block's execution with dynamic decision variables. In the context of pipelines, [HB84] presented a similar structure to dynamically reconfigure the initiation rate for multi-function pipelines.

In the following section, we will show how CFEs and decision variables can be used to represent the design constraints.

3.2 Constraint Specification

Constraints are properties that any implementation needs to satisfy. In general, the system being synthesized needs to communicate with the external environment, or with custom made components, for which an interface has been specified. The interface properties must be satisfied in all valid implementations for the system to work properly. In addition to interface constraints, the user may impose structural constraints that will limit the degrees of freedom used to implement a behavior. For example, the user may limit a design to use a single bus for communication with the other models.

Since specifying the environment under which a system is going to execute is formidable task, several approaches have been proposed in the past to specify the system's constraints. In [NT86, KM92, Bor88, HLH91, RB93], the environment's constraints were specified as minimum or maximum timing constraints between two operations. Since the algorithms for finding schedules satisfying those timing constraints were limited to basic blocks, these timing constraints were limited to operations in the same basic block. This prevented designers from specifying more complex interactions existing in control-dominated specifications, such as timing constraints in alternative paths or loops.

More recently [TWL95], the environment's timing constraints have been extended to allow minimum and maximum timing constraints on arbitrary paths in the specification, and in [YW] an algorithm was proposed that obtains schedules satisfying these timing constraints. Using this new formulation, for example, the user can constrain the execution time for the hit-case of a cache controller, while using a different timing constraint for the miss-case of the same cache controller. Note however, that since the hit and miss cases belong to the same description and share parts of the specification, these shared parts will be constrained by the hit and miss timing constraints.

The set of timing constraints specifying the interactions between the system and its environment can become unmanageable in some designs because of the number of interactions among the different parts, as in the case of a model whose environment is a bus protocol or in the case we want to synthesize protocol converters between two specifications. Thus, we need to extend the methods for specifying constraints presented above by allowing the system to be described as a parallel composition of the specification and a suitable representation of the environment.

Figure 18 depicts the representation a process P corresponding to a circuit specification, and its environment, represented by process Env. Note that all timing constraints are encapsulated by process Env which synchronizes with the specification for P. Thus, the system under consideration for synthesis purposes is represented by the CFE $p||e$, where p is the control-flow expression for P and e is a CFE representing a suitable model for P's environment.

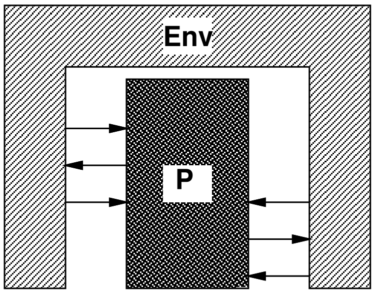


Figure 18: *Process P and its Environment*

Note that we do not restrict our discussion to timing constraints in the specification of the environment, but we also allow the environment to contain resource usage constraints and synchronization constraints. These types of constraints are used in this section as building blocks to construct more complex constraints with control-flow expressions.

The reader must keep in mind the the goals in the synthesis flow. We want to solve two problems: statically scheduling the operations inside basic blocks using these system's constraints, and dynamically synchronizing the parts for which no static schedule can be found. This yields two types of constraints that will be handled differently during synthesis for efficiency reasons: statically satisfiable constraints which constrain the operations inside a basic block, and dynamically satisfiable constraints, which require the synthesis of synchronization skeletons for the process being synthesized.

We will first present dynamically satisfiable constraints, which uses compositions of control-flow expressions. Then, we will define statically satisfiable constraints as a subset of dynamically satisfiable constraints.

3.2.1 Dynamically Satisfiable Constraints

In this section, we begin by presenting the specification of constraints that can be dynamically satisfiable. 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 are defined in terms of control-flow expressions. In binding constraints, we use expression rewriting, i.e., the incorporation of binding constraints as a modification of the original CFE. Both timing and binding constraints use decision variables as quantifiers of the design space. Finally, synchronization constraints use multisets of actions that should occur at the same time and multisets of actions that should never occur at the same time.

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

Definition 3.3 *The support of a control-flow expression p is defined as the set of actions that are executed in p.*

> **Example 3.2.3.** 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, $S_p = \{a, b, c, d, e\}$. □

Associated with each action defined in the support of a CFE, we have a shadow action, which executes every time the corresponding action executes.

Definition 3.4 *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 3.2.4.** In the CFE $(a \cdot b \cdot c)^\omega$, σ_a, σ_b and σ_c execute every time a, b and c are executed, respectively. □

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 [TWL95, WTHM92, KM92, Bor88]. The specification of scheduling constraints using control-flow expressions can be also considered as an extension of path constraints defined by [TWL95].

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 3.5 *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.*

In this definition for minimum timing constraint, the CFE waits until action a executes by setting x to *true* for multiple cycles. Then, it counts $n-1$ cycles (0^{n-1}) and waits for action b to execute, which represented by b's shadow action σ_b.

Definition 3.6 *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.*

The constraint defined above defines a maximum time between two actions by first waiting for action a to execute, represented by its shadow action, and then waiting at most $n-1$ cycles until it allows action b to execute, by synchronizing with b's shadow action.

The following theorem shows how to combine the two definitions in order to obtain a timing constraint defined as the delay between two actions.

Theorem 3.1 *Let* $\texttt{min} = (x_1 : 0)^* \cdot \sigma_a \cdot 0^{n-1} \cdot (x_2 : 0)^* \cdot \sigma_b$ *be a minimum timing constraint of* n *cycles between* a *and* b*, and let* $\texttt{max} = (x_3 : 0)^* \cdot \sigma_a \cdot (x_4 : 0)^{<n} \cdot \sigma_b$ *be a maximum timing constraint of* n *cycles between* a *and* b*, where* x_1*,* x_2*,* x_3 *and* x_4 *are decision variables. The CFE* $\texttt{min}||\texttt{max}$ *is the timing constraint representing exactly* n *cycles between* a *and* b*, which can be rewritten as* $(y : 0)^* \cdot \sigma_a \cdot 0^{n-1} \cdot \sigma_b$*, where* y *is a decision variable.*

> **Proof:** Note that x_1 and x_3 will have the same assignment over time, since they will be synchronizing with the execution of σ_a, and this assignment will be the same as the assignments over time for y for the same reason.
>
> Because the execution of b in both constraints is synchronized to the execution of the respective shadow actions, x_2 will always be assigned to 0, and x_4 will be assigned 1 in the first $n-1$ cycles, and then 0. If we substitute these assignments to x_3 and x_4, we obtain $(y : 0)^* \cdot \sigma_a \cdot 0^{n-1} \cdot \sigma_b$. $\sharp$

We can now define the representation of a system and the environment specified in terms of timing constraints and the auxiliary actions.

Definition 3.7 *Let* p *be a control-flow expression modeling a system, and let* $m_1, \ldots,$ m_n *be a set of CFEs representing scheduling constraints from the environment. The control-flow expression* $p||m_1|| \cdots ||m_n$ *denote the system consisting of* p *and its environmental constraints. We call this new expression the application of timing constraints to* p*.*

> **Example 3.2.5.** In the differential equation solver of Example 2.3.1, assume the CFE for the specification is p, and that we want to specify a maximum timing constraint of 3 cycles between m_4 and s_2, which can be represented by the CFE $(x_1 : 0)^* \cdot \sigma_{m_4} \cdot (x_2 : 0)^{<3} \cdot \sigma_{s_2}$, where x_1 and x_2 are decision variables.
>
> The application of this constraint to the CFE p is represented by a new CFE $p||(x_1 : 0)^* \cdot \sigma_{m_4} \cdot (x_2 : 0)^{<3} \cdot \sigma_{s_2}$. $\Box$

In the previous example, we specified conventional minimum and maximum timing constraints. As we pointed out before, CFEs 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 3.2.6.** In the differential equation solver of Example 2.3.1, let p be the process representing the CFE for the basic block. For the CFE $q = ((x_1 : 0)^* \cdot a \cdot p)^\omega || ((x_2 : 0)^* \cdot b \cdot p)^\omega$, the constraint $(x : 0)^* \cdot \sigma_a \cdot (c : 0^2 + \overline{c} : 0^3) \cdot \sigma_b$ indicates that the first basic block should execute either 2 cycles or 3 cycles before the second one, depending on the conditional c. □

Binding Constraints

Binding constraints specify the possible implementations for each computation that is represented by an action. We represent binding constraints as a rewriting of the original control-flow expression.

Definition 3.8 *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 3.2.7.** 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 3.9 *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:*

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

where $\sum_{1 \leq i \leq m} x_i : C_i$ 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 0 or 1, 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. Note also that if some C_i executes in more than one cycle, then the timing semantics of the CFE will be changed.

Example 3.2.8. In this example, let us assume that actions $m_i, i = 1, \ldots, 5$ of Figure 17 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 1 to 5 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.

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 grouped multisets of actions.

Definition 3.10 *Let ALWAYS be a set consisting of multisets of actions that contain 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 3.11 *Let NEVER be a set consisting of multisets of actions that contain 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 3.2.9. Let us consider the synchronization synthesis problem presented in Chapter 1. In this problem, let us assume the following control-flow expressions for the processes *DMArcvd, DMAxmit* and *enqueue*, respectively:

$$\begin{aligned} p_1 &= [a.0]^\omega \\ p_2 &= [0.(c:0)^*.a]^\omega \\ p_3 &= [(x:0)^*.a]^\omega \end{aligned}$$

Where a corresponds to the bus access and 0 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 $NEVER = \{\{a, a\}\}$. □

If grouped, the sets of multisets of actions represent alternative synchronizations. For example, the multiset of actions $\{\{a, d\}, \{b, d\}, \{c, d\}\}$ can be represented by $\{\{\{a, b, c\}, d\}\}$. If this multiset is contained in the *ALWAYS* set, then d must execute when any of a, b or c execute.

3.2.2 Statically Satisfiable Constraints

In this section, we consider the specification of statically satisfiable constraints, i.e., constraints that will be satisfied by a constant assignment of values to the decision variables inside the system's basic blocks.

Because these constraints will be treated differently than the constraints specified before, we will use a different notation for specifying them. Instead of being compositional, statically satisfiable constraints will be specified explicitly as path-activated constraints, resource limiting constraints and environment processes.

During synthesis, statically satisfiable constraints will determine static relationships between the decision variables in basic blocks. As a result, the notation for specifying them will be much more restricted than the compositional approach we presented in the previous subsection. In particular, we cannot allow constraints to introduce any new conditional to the CFE representing the system.

We consider in this section three types of constraints: path-activated constraints, resource constraints, which is a limited form of synchronization constraints, and environment processes, which are processes that statically constrain the execution of a control-flow expression. We will be mostly interested in the syntax of these statically satisfiable constraints, and in the following chapters, we will be considering the semantics of them.

Path-Activated Constraints

Path-activated constraints for CFEs can be considered as an extension to the path-activated constraints defined in [YW], where they were used to specify global timing constraints crossing the alternative and sequential compositions of the specification. In CFEs, we will also allow these constraints to cross parallel and exception handling compositions, which were defined in the previous chapter. These path-activated constraints will uniquely determine relationships for the decision variables of the system being synthesized.

In a path-activated constraint, we specify actions and conditionals that must be executed in a given order, but not necessarily form a sequence (where the execution of one action or conditional follows immediately that of its predecessor action or conditional). We call this ordered set a *thread*.

Definition 3.12 *A thread of a CFE p is the ordered set $[l_1, l_2, \ldots, l_n]$, where l_i is an action in the support of p, or a guard without decision variables. In a thread, l_i cannot execute after l_{i+1}, $i \in \{0, \ldots, n-1\}$.*

We say that a thread has been executed if the corresponding CFE p executes all the corresponding actions and conditionals.

A path-activated constraint specify a limit on the number of cycles spent by the CFE to execute a thread.

Definition 3.13 *Let p be an CFE and let $[l_1, l_2, \ldots, l_n]$ be a thread. Then,* $\mathbf{min}(t, [l_1, l_2, \ldots, l_n])$*,* $\mathbf{max}(t, [l_1, l_2, \ldots, l_n])$ *and* $\mathbf{delay}(t, [l_1, l_2, \ldots, l_n])$ *are path-activated constraints denoting the minimum, maximum and exact delay of t cycles allowed for the execution of the thread.*

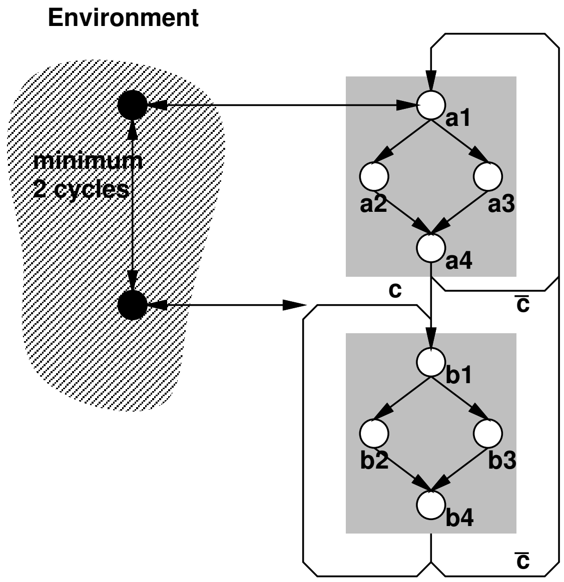


Figure 19: *Path-activated constraint*

Example 3.2.10. In Figure 19, we present pictorial view of the control-flow expression $(\{a_1 \to a_2, a_1 \to a_3, a_2 \to a_4, a_3 \to a_4\} \cdot (c : \{b_1 \to b_2, b_1 \to b_3, b_2 \to b_4, b_3 \to b_4\})^*)^\omega$, with the environment requiring a_1 be executed at least two cycles apart from the evaluation of c.

Since this constraint will only constrain the possible execution times for the operations inside the basic blocks of the system, we will represent this as the path-activated constraint $\mathbf{min}(2, [a_1, c])$. $\square$

Resource Usage Constraints

We introduce here resource usage constraints for limiting the number of concurrent executions of a set of actions.

Definition 3.14 *Let $A = \{a_1, \ldots, a_m\}$ be a set of actions, and let n be maximum number of concurrent executions of the actions in A. Then the constraint* $\mathbf{limit}(n, A)$ *denotes the maximum number of concurrent executions of the actions in A.*

> **Example 3.2.11.** In the differential equation basic block of Example 3.2.1, we can limit the number of multipliers m_i executing concurrently by specifying the constraint $\mathbf{limit}(1, \{m_1, m_2, m_3, m_4, m_5, m_6\})$.
>
> Note that we could have represented this constraint by the synchronization constraint $NEVER = \cup_{i,j=1\ldots 6, i \neq j}\{m_i, m_j\}$. However, as mentioned before, we want to distinguish constraints that can be satisfied statically from constraints that can be dynamically satisfiable, and the definition above will be more convenient. □

Environment Processes

Path-activated constraints can be hard to specify if their number is large. In these cases, it may be easier to constrain the execution of p by a process executing concurrently with p that encapsulates the path-activated constraints. This process must use a suitable sub-set of CFEs, since the specification power of CFEs may lead to constraints that cannot be statically satisfied.

- The environment process must not introduce any new conditionals in the specification, since this may require the combined specification to have different schedules for the different conditionals.

- If two actions a and b of process p synchronize with the actions $\underline{a}$ and $\underline{b}$ of p's environment, then every path-activated constraint that can be written from $\underline{a}$ to $\underline{b}$ must define the same delay between $\underline{a}$ and $\underline{b}$ if they correspond to the same path-activated constraint in p, when considering the *ALWAYS* set containing $\{\{a, \underline{a}\}, \{b, \underline{b}\}\}$.

Although these requirements are not necessary conditions for p's environment to yield statically satisfiable assignments to the decision variables, they will make p and p's environment tightly coupled, and we will be able to extract the constraints for p's static decision variables from this tightly couple specification.

```
fork : BLK_0
   begin : BLK_00
      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 BLK_0;
   end
   begin : BLK_01
      wait (posedge CCT);
      disable BLK_0;
   end
join
```

Figure 20: *Exception handling in Verilog HDL*

Example 3.2.12. We repeat in Figure 20 the original Verilog description of Example 2.4.7, and we rewrite processes p_{00} and p_{01} below. These processes correspond to the blocks *BLK_00* and *BLK_01* of Figure 20, respectively.

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

In this example, we assume that each o_i represents the process $(x_i : 0)^* \cdot out_i$ that first waits until the assignment can be performed. This assignment is constrained by another process that converts the bytes transmitted from variable `out` into bits. As a result, every assignment to `out` must be eight cycles apart. Instead of specifying the possible timing constraints between every output assignment, we can specify the environment process *env*, which composes in parallel with p and constrains its execution. This environment process is presented below.

$$\begin{aligned} p &= (p_{00} \| p_{01} \| env) \\ env &= (o_r \cdot 0^7)^\omega \end{aligned}$$

Process p contains the *ALWAYS* set $\{\{\{out_1, \ldots, out_9\}, o_r\}\}$. □

3.3 Summary

In this chapter, we showed how the environmental constraints of a design were abstracted by CFEs. Since designs at the higher levels of abstraction were non-deterministic due to the multiple design choices, we encapsulated these design choices in CFEs by guarding them with decision variables. Decision variables allowed us to uniquely identify the different design choices that were necessary for quantifying a design.

We defined two types of decision variables, static decision variables, which were used to guard the execution of actions in basic blocks, and dynamic decision variables, which could to be used anywhere in the CFE to model a synchronization point or a design choice that could admit multiple choices.

We partitioned the specification of design constraints into dynamically satisfiable constraints and statically satisfiable constraints. Dynamically satisfiable constraints were defined in such way that they modified the original specification of the system. Timing constraints were composed in parallel with the specification. Resource binding constraints were introduced by expression rewriting, and synchronization constraints specified actions that could execute at the same time and actions that could not execute at the same time. We also showed that these constraints could be used as building blocks to represent more complex interactions between the specification and its environment.

Statically satisfiable constraints were used to constrain the decision variables inside basic blocks. Because these constraints would be satisfiable for all possible executions of an implementation, we restricted the constraint not to change the flow of control in the original specification. This led to path-activated constraints, which were a general form for minimum and maximum timing constraints; action limiting constraints, that limited the number of concurrent actions executing concurrently; and environment processes, which were processes tightly coupled with the specification whose sole purpose was to constrain the possible static assignments to the decision variables.

Chapter 4

Analysis of Concurrent Systems

In order to synthesize controllers satisfying environmental constraints, we have to analyze the (extended) control-flow expression representing the system to identify the regions of computation that execute either concurrently or exclusively. These regions will help us to obtain tighter constraints among the different blocks of the specification. In this chapter, we analyze the CFE representing a system by constructing a specification automaton with the same behavior as the control-flow expression model. The construction of the specification automaton uses the notion of a *derivative* of a control-flow expression, which is a one-cycle simulation of the behavior of the CFE. This specification automaton enables us to check the feasibility of implementations subject to the constraints of the design.

The outline of this chapter is as follows. In Section 4.1, we first present the definition of derivatives for the control-flow expressions defined in Section 2.2, followed by the definition of derivatives for extended control-flow expressions. Then, in Section 4.2, we present an algorithm for constructing the finite-state machine representation for an ECFE, and in Section 4.3 we address some implementation issues regarding the representation of the finite-state machine as a transition relation.

4.0 Notation

In this thesis, we will use the following notation. If f and g are Boolean functions, fg and $f \vee g$ correspond to their conjunction and disjunction respectively, while $\overline{f}$ denotes the complementation of f (i.e. the symbol $\wedge$ is dropped for the sake of simplicity). For a set of Boolean functions f_i, $i \in \{1, \ldots, n\}$, $\bigwedge_i f_i$ and $\bigvee_i f_i$ represent the conjunction and the disjunction of all f_i's, respectively.

If p_i, $i \in \{1, \ldots, n\}$ are functions from a set $\mathcal{S}$ to integer numbers, then $p_1 + p_2$ represent the arithmetic sum of p_1 and p_2, and $\sum_i p_i$ denotes the arithmetic sum of all p_i's.

4.1 Control-Flow Finite State Machines

This section shows how to generate a finite-state representation called a *control-flow finite state machine* — or CFFSM — from control-flow expressions. We use both the algebraic and the finite-state representations for analysis and synthesis, as well as in our implementation tool *Thalia*. The algebraic representation presented in the previous chapters allows us to manipulate and rewrite the expressions algebraically, as in the case when we introduced environment processes to constrain the specification. 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 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 [Brz64] 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 4.1.1. 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 Figure 21.

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. □

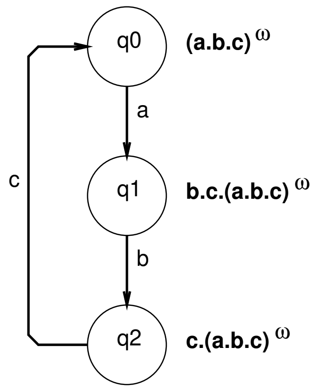


Figure 21: *Mealy machine for control-flow expression* $(a \cdot b \cdot c)^\omega$

What we need to show now is how to compute the suffixes of a control-flow expression, that there are only a finite number of suffixes for a given CFE, and that there is an equivalence relation between the set of suffixes of a control-flow expression and the states of its corresponding Mealy automaton. In the next section, we will present the definition of derivatives for the control-flow expressions that were originally defined in

Section 2.2. Then, in the following section, we will extend the definition of derivatives to include extended control-flow expressions.

4.1.1 Derivatives of Control-Flow Expressions

We show in this section 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 the derivative of a control-flow expression, we first need to know when the CFE executes in zero time.

Execution in Zero Time

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. The definition of Δ is given below.

Definition 4.1 *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. Let also $c_1, c_2, g \in \mathcal{G}$.*

   $\Delta(p \cdot q) = \Delta(p)\,\Delta\,(q)$

   $\Delta(c_1 : p + c_2 : q) = c_1\,\Delta\,(p) \vee c_2\,\Delta\,(q)$

   $\Delta((g : p)^*) = \overline{g}$

   $\Delta(p^\omega) = 0$

   $\Delta(p||q) = \Delta(p)\,\Delta\,(q)$

Example 4.1.2. Let $p = (c : a + \overline{c} : (d : e \cdot f)^*)$. Then $\Delta(p) = \overline{c}\overline{d}$. □

The function Δ determines which assignment to conditionals and decision variables makes a control-flow expression execute ϵ, which 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^ω), $(c : p)^*$ 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 4.2 *Let p be a control-flow expression. We say $(c : p)^*$ and p^ω are well-formed CFEs (WFCFEs) if $\Delta(p) = 0$.*

Although non-WFCFEs 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 non-WFCFEs into WFCFEs such that the execution time for the non-ϵ executions is maintained, and a delay is generated for ϵ executions.

Theorem 4.1 *Let $\Delta(p) \neq 0$, for some CFE p. Then $\Delta(p \cdot (\Delta(p) : 0 + \overline{\Delta(p)} : \epsilon)) = 0$.*

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

Example 4.1.3. Let $p = (c : a + \overline{c} : (d : e \cdot f)^*)$, whose $\Delta(p) = \overline{c}\overline{d}$. The CFE $q = p^\omega$ is not well formed, since when $\overline{c}\overline{d}$ is *true*, a loop executing in 0-cycles occurs. We can convert q into the expression $q = (p \cdot (\overline{c}\overline{d} : 0 + \overline{\overline{c}\overline{d}} : \epsilon))^\omega$, which contains the same behavior as q for the executions that take one or more cycles, and executes in one cycle whenever p would execute in zero cycles. □

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 $\mathcal{G} \times \mathcal{M}^{\mathcal{A}} \times \mathcal{F}$, 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 4.3 *Let $\partial : \mathcal{F} \rightarrow \mathcal{G} \times \mathcal{M}^{\mathcal{A}} \times \mathcal{F}$ be the a derivative of a control-flow expression, given recursively as follows:*

$$
\begin{aligned}
\partial f : \epsilon &= \{(f, \epsilon, \epsilon)\} \\
\partial f : a &= \{(f, a, \epsilon)\} \\
\partial \delta &= \emptyset \text{ , the empty set} \\
\partial f : 0 &= \{(f, 0, \epsilon)\} \\
\partial p \cdot q &= \{(\gamma, \mu, \pi \cdot q) \mid (\gamma, \mu, \pi) \in \partial p \;\; \mathit{for} \;\; (\mu \neq \epsilon)\} \cup \\
&\quad\ \{(\Delta(p)\gamma, \mu, \pi) \mid (\gamma, \mu, \pi) \in \partial q\} \\
\partial f_1 : p + f_2 : q &= \{(f_1\gamma, \mu, \pi) \mid (\gamma, \mu, \pi) \in \partial p \;\; \mathit{for} \;\; (\mu \neq \epsilon)\} \cup \\
&\quad\ \{(f_2\gamma, \mu, \pi) \mid (\gamma, \mu, \pi) \in \partial q \;\; \mathit{for} \;\; (\mu \neq \epsilon)\} \\
\partial p^\omega &= \{(\gamma, \mu, \pi) \mid (\gamma, \mu, \pi) \in \partial(p \cdot p^\omega)\} \\
\partial (f : p)^* &= \{(f\gamma, \mu, \pi) \mid (\gamma, \mu, \pi) \in \partial(p \cdot (f : p)^*)\} \\
\partial p \| q &= \{(\gamma_p\gamma_q, \mu_p \cup \mu_q, \pi_p \| \pi_q) \mid (\gamma_p, \mu_p, \pi_p) \in \partial p \;\; \mathit{and} \\
&\quad\ (\gamma_q, \mu_q, \pi_q) \in \partial q\}
\end{aligned}
$$

Example 4.1.4. Let $p = (a \cdot b \cdot c)^\omega$.

$$
\begin{aligned}
\partial p &= \{(\gamma, \mu, \pi) \mid (\gamma, \mu, \pi) \in \partial(a \cdot b \cdot c \cdot (a \cdot b \cdot c)^\omega\} \\
&= \{(\gamma, \mu, \pi \cdot b \cdot c \cdot (a \cdot b \cdot c)^\omega) \mid (\gamma, \mu, \pi) \in \partial a\} \cup \emptyset \\
&= \{(\mathit{true}, a, b \cdot c \cdot (a \cdot b \cdot c)^\omega)\}
\end{aligned}
$$

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

4.1.2 Derivatives in Extended Control-Flow Expressions

In this section, we extend the computation of derivatives presented in the previous section to the ECFEs defined in Section 2.4.

Since we presented new compositions for ECFEs, we have to show first that there exist equivalent CFEs for exception handling and basic blocks. Registers require us to extend derivatives to consider the values that a register may have over time.

Preliminaries

We establish here the preliminary definitions that will be necessary in order to compute the derivatives of ECFEs. In particular, we adapt the definitions presented in the previous chapters for CFEs. We use an alternative definition for the disable construct, present the definition of a memory in order to analyze CFEs with registers, determine a CFE equivalent to a basic block, and check if a CFE can execute in zero time.

Exception Handling: Instead of using the definition for the disable $\zeta(n,p)$ that we defined in Chapter 2, we will use in this chapter the definition of a more general exception handling mechanism, named $\overline{\zeta}(q)$, which "forgets" the original CFE and executes q instead. Later in this chapter, we show that $\exists q$ for all $\zeta(n,p)$ such that $\zeta(n,p) = \overline{\zeta}(q)$.

Example 4.1.5. Let $p = (a \cdot b \cdot (c_1 : \zeta(2,p) \cdot c + \overline{c}_1 : \epsilon) \cdot d)^\omega$. Then, $\zeta(2,p)$ can be replaced by $\overline{\zeta}(d \cdot p)$, since $d \cdot p$ is the CFE that follows the execution of $\zeta(2,p)$, the disabling of $(c_1 : \zeta(2,p) \cdot c + \overline{c_1} : \epsilon)$. □

Registers: In order to define the behavior of register variables, let us define the storage of a CFE. Let $\mathcal{R}$ be a set of registers for a CFE p, r one of the registers in $\mathcal{R}$ and $|r|$ the number of possible values for r.

Definition 4.4 *The storage ξ of p is a partial mapping $\xi : \mathcal{R} \rightarrow \{0, \ldots, |r|-1\}$ i.e., it associates a number in $\{0, \ldots, |r|-1\}$ with some of registers $r \in \mathcal{R}$.*

We represent the different partial functions of $\mathcal{R}$ by ξ, ξ', ξ'', etc. In particular, ξ_0 denotes the total mapping in which all registers $r \in \mathcal{R}$ have a value zero ($\xi(r) = 0$). We can view the storage as a vector, whose entries are the values (when assigned) of the register variables. Thus ξ_0 is a vector whose entries are all 0s. We use the notion of *derived storage* ($\xi \bullet [r \leftarrow n]$) to assign values to the vector entries of ξ, representing the loading of the corresponding register. Namely:

$$\xi \bullet [r \leftarrow n](x) = \begin{cases} \xi(x) & \text{if } x \neq r \\ n & \text{otherwise} \end{cases}$$

where n is an integer modulus $|r|$.

The storage of a CFE represents the memory associated with the registers and it will be part of the state of the system modeled by a CFE. In general, we will refer to derived storages as $\xi \bullet \xi'$ when the registers that are defined in ξ' are irrelevant or unknown. We denote by $\mathcal{S}$ the set of all mappings $\mathcal{R} \rightarrow \{0, \ldots, |r|-1\}$.

Basic Blocks: The behavior of a basic block is defined in terms of the possible schedules for the operations in the basic block. We mentioned before that the definition of a basic block was just a convenience for efficiently capturing the structure of basic blocks from programming languages and hardware description languages. We will show in the sequel that basic blocks can be represented by a CFE without basic blocks, once we determine the possible schedules for the operations in the basic block. Let us assume a basic block can be executed in at most s cycles ($\{1,\ldots,s\}$) and for each cycle $i \in \{1,\ldots,s\}$, let us define the set X_i of decision variables such that if $x \in X_i$, there is an action that can be scheduled at cycle i if x is *true*. Since in Chapter 2 we also defined a decision variable for each cycle of the basic block, we will call this decision variable y_i.

Because the basic block has not been scheduled yet, we have to compute when a cycle i of the basic block will be used in the finite-state machine representation of the CFE. Informally, we say that a basic block executes up to cycle i if there is an operation scheduled at cycle i (i.e., if there is $x \in X_i$ that is assigned to *true*), if the decision variable y_i corresponding to cycle i is *true*, or if the basic block needs to execute the following clock cycle. More formally, executing a basic block up to cycle i can be computed by the following recursive definition.

Definition 4.5 *Let s be the maximum number of cycles of a basic block p, and let* $\mathrm{et}(p,i)$ *be a Boolean function computing the conditions under which p executes in at least i cycles. This function can be represented in terms of decision variables, as shown below:*

$$\begin{array}{rcll} \mathrm{et}(p,i) & = & 0 & \textit{if } i > s \\ \mathrm{et}(p,i) & = & \bigvee\limits_{x \in X_i} x \ \vee \ \mathrm{et}(p,i+1) \ \vee \ y_i & \textit{if } i \in \{1,\ldots,s\} \end{array}$$

We can now define a CFE which is equivalent to a basic block p. This CFE will represent the cycle by cycle behavior of the basic block. In each cycle, the actions that can be executed will be guarded by the corresponding decision variable. The execution of the following cycle of the basic block will be guarded by the function et described above.

Definition 4.6 *Let p be a basic block whose actions are a_i. Let the possible cycles for the basic block be $\{1,\ldots,s\}$, and let each action a_i have decision* *variables x_{ij}, according to the basic block's precedence constraints such that if x_{ij} is true, then action a_i is scheduled in cycle j.*

Then $p \equiv p_1$, where

$$
\begin{aligned}
p_j &= (\mathrm{et}(p,j) : (\|_Q(x_{ij} : a_i + \overline{x_{ij}} : 0)) \cdot p_{j+1} + \overline{\mathrm{et}}(p,j) : \epsilon) \\
&\quad \textit{for } j \in \{1, \ldots, s\} \\
p_{s+1} &= \epsilon
\end{aligned}
$$

where Q is the set of actions a_i such that $x_{ij} \in Xj$, and $\|_Q(x_{ij} : a_i + \overline{x_{ij}} : 0)$ is the parallel composition $(x_{1j} : a_1 + \overline{x_{1j}} : 0)\| \ldots \|(x_{nj} : a_n + \overline{x_{nj}} : 0)$.

Example 4.1.6. Let $p = \{o_1 \to o_2, o_1 \to o_3\}$ be a basic block. Let us assume that decision variables $\{x_{11}, x_{12}\}$, $\{x_{22}, x_{23}\}$, $\{x_{32}, x_{33}\}$ define when o_1, o_2 and o_3 can execute, respectively. For example, o_1 can be executed in the first cycle of the basic block if x_{11} is *true*, and it can be executed in the second cycle of the basic block if x_{12} is *true*. We also assume that y_1, y_2 and y_3 are decision variables that are *true* if the first, second or third cycles of the basic block needs to be executed, respectively.

We can define the conditions when each of the cycles of the basic block will be executed as follows:

$$
\begin{aligned}
\mathrm{et}(p,3) &= y_3 \vee x_{23} \vee x_{33} \\
\mathrm{et}(p,2) &= \mathrm{et}(p,3) \vee y_2 \vee x_{12} \vee x_{22} \vee x_{32} \\
\mathrm{et}(p,1) &= \mathrm{et}(p,2) \vee y_1 \vee x_{11}
\end{aligned}
$$

The CFE p_1 defined below will be equivalent to the basic-block p.

$$
\begin{aligned}
p_1 &= (\mathrm{et}(p,1) : (x_{11} : o_1 + \overline{x_{11}} : 0) \cdot p_2 + \overline{\mathrm{et}}(p,1) : \epsilon) \\
p_2 &= (\mathrm{et}(p,2) : ((x_{12} : o_1 + \overline{x_{12}} : 0)\|(x_{22} : o_2 + \overline{x_{22}} : 0)\| \\
&\quad (x_{32} : o_3 + \overline{x_{32}} : 0)) \cdot p_3 + \overline{\mathrm{et}}(p,2) : \epsilon) \\
p_3 &= (\mathrm{et}(p,3) : ((x_{23} : o_2 + \overline{x_{23}} : 0)\|(x_{33} : o_3 + \overline{x_{33}} : 0)) \cdot p_4 + \\
&\quad \overline{\mathrm{et}}(p,3) : \epsilon) \\
p_4 &= \epsilon
\end{aligned}
$$

In this CFE representation of the basic block, if $\mathrm{et}(p, 1)$ is *true*, then p_1 executes o_1 in the first cycle if x_{11} is *true*, or it waits one cycle if x_{11} is *false*. Then, p_1 transfers execution to p_2. If $\mathrm{et}(p, 1)$ is *false*, then p_1 aborts the execution of the basic block through an ϵ-path. □

Execution in Zero Time: We defined in the previous section the function Δ which returns a Boolean guard enabling zero-cycle paths in a CFE. Here, we extend the Δ function defined previously to include basic blocks, exception handling and registers.

Definition 4.7 *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. Let also v be a register of a control-flow expression, and let r_i be a set of precedence constraints.*

$$\Delta(p \cdot q) = \Delta(p)\, \Delta\,(q)$$

$$\Delta(c_1 : p + c_2 : q) = c_1\, \Delta\,(p) \vee c_2\, \Delta\,(q)$$

$$\Delta((g : p)^*) = \overline{g}$$

$$\Delta(p^\omega) = 0$$

$$\Delta(p || q) = \Delta(p)\, \Delta\,(q)$$

$$\Delta(\overline{\zeta}(p))^1 = \begin{cases} \Delta(p) & \textit{if } \Delta(p) \textit{ do not depend on } \Delta(\overline{\zeta}(p)) \\ 1 & \textit{otherwise} \end{cases}$$

$$\Delta([v \leftarrow \textit{constant}]) = 0$$

$$\Delta([v + +]) = 0$$

$$\Delta([v - -]) = 0$$

$$\Delta(\{r_1, \ldots, r_n\}) = 0$$

Example 4.1.7. Let $p = ((c_1 : \overline{\zeta}(d \cdot p) + \overline{c_1} : 0) \cdot d)^\omega$. Then,

$$\begin{aligned} \Delta(p) &= c_1 \ \Delta\left(\overline{\zeta}(d \cdot p)\right) \\ &= c_1 \ \Delta\left(d \cdot p\right) \\ &= 0 \end{aligned}$$

If $p = ((c_1 : \overline{\zeta}(p) + \overline{c_1} : 0) \cdot d)^\omega$, then,

$$\begin{aligned} \Delta(p) &= c_1 \ \Delta\left(\overline{\zeta}(p)\right) \\ &= c_1 \end{aligned}$$

□

The definition of well-formed CFEs presented previously and Theorem 4.1 also applies to CFEs containing basic blocks, exception handling and register variables.

Example 4.1.8. In the CFE $p = (c_1 : \overline{\zeta}(p) + \overline{c_1} : 0)^\omega$, since $(c_1 : \overline{\zeta}(p) + \overline{c_1} : 0)$ is not well-formed in p, we have to modify the unconditional repetition, which results in the CFE $((c_1 : \overline{\zeta}(0 \cdot p) + \overline{c_1} : 0) \cdot (c_1 : 0 + \overline{c_1} : \epsilon))^\omega$.
□

We can now extend derivatives of control-flow expressions. Because the cycle-by-cycle evolution of a CFE depends not only on the CFE, but also on the possible values a register may have, the derivative of a CFE will have to consider storages and derived storages. The operator ∂ applied to a CFE and a storage, yields the quadruple $\mathcal{G} \times \mathcal{M}^{\mathcal{A}} \times \mathcal{F} \times \mathcal{S}$, where $\mathcal{G}$ is the set of Boolean expressions over the set of

[1] In this case, if in order to compute $\Delta(p)$ we need to know the result of $\overline{\zeta}(p)$, an ϵ-path exists in $\overline{\zeta}(p)$. Thus, we signal that by setting $\Delta(\overline{\zeta}(p))$ to 1.

conditional and decision variables, $\mathcal{M}^{\mathcal{A}}$ is the set consisting of multisets of actions, $\mathcal{F}$ is the set of control-flow expressions, and $\mathcal{S}$ is the next storage. The quadruple $(\gamma, \mu, \pi, \xi') \in \mathcal{G} \times \mathcal{M}^{\mathcal{A}} \times \mathcal{F} \times \mathcal{S}$ obtained from a CFE p and storage ξ indicates that the actions μ are executed when γ is *true*, followed by the execution of π, and yielding the derived storage ξ' from ξ. We will omit the storage from the derivative definition whenever no register variables were defined for a CFE.

Definition 4.8 *Let $\partial : \mathcal{F} \times \mathcal{S} \rightarrow \mathcal{G} \times \mathcal{M}^{\mathcal{A}} \times \mathcal{F} \times \mathcal{S}$ be defined as a derivative of a control-flow expression, given recursively as follows:*

$$
\begin{array}{rcl}
\partial(f : \epsilon, \xi) & = & \{(f, \epsilon, \epsilon, \xi)\} \\
\partial(f : a, \xi) & = & \{(f, a, \epsilon, \xi)\} \\
\partial(\delta, \xi) & = & \emptyset \text{ , the empty set} \\
\partial(f : 0, \xi) & = & \{(f, 0, \epsilon, \xi)\} \\
\partial(\overline{\zeta}(p), \xi) & = & \partial(p, \xi) \\
\partial(f :\ [r++], \xi) & = & \{(f, 0, \epsilon, \xi \bullet [r \leftarrow \xi(r) + 1 \bmod |r|]\} \\
\partial(f :\ [r--], \xi) & = & \{(f, 0, \epsilon, \xi \bullet [r \leftarrow \xi(r) - 1 \bmod |r|]\} \\
\partial(f :\ [r := n], \xi) & = & \{(f, 0, \epsilon, \xi \bullet [r \leftarrow n \bmod |r|]\} \\
\partial(\{r_1, \ldots, r_n\}, \xi) & = & \partial(p_1, \xi) \text{ , where } p_1 \text{ is the CFE corresponding to the basic} \\
 & & \text{block, defined previously in this section.} \\
\partial(p \cdot q, \xi) & = & \{(\gamma, \mu, \pi \cdot q, \xi') \mid (\gamma, \mu, \pi, \xi') \in \partial(p, \xi) \ \textit{for} \ (\mu \neq \epsilon)\} \cup \\
 & & \{(\Delta(p)\gamma, \mu, \pi, \xi'') \mid (\gamma, \mu, \pi, \xi'') \in \partial(q, \xi)\} \\
\partial(f_1 : p + f_2 : q, \xi) & = & \{(f_1\gamma, \mu, \pi, \xi') \mid (\gamma, \mu, \pi, \xi') \in \partial(p, \xi) \ \textit{for} \ (\mu \neq \epsilon)\} \cup \\
 & & \{(f_2\gamma, \mu, \pi, \xi'') \mid (\gamma, \mu, \pi, \xi'') \in \partial(q, \xi'') \ \textit{for} \ (\mu \neq \epsilon)\} \\
\partial(p^\omega, \xi) & = & \{(\gamma, \mu, \pi, \xi') \mid (\gamma, \mu, \pi, \xi') \in \partial(p \cdot p^\omega, \xi)\} \\
\partial((f : p)^*, \xi) & = & \{(f\gamma, \mu, \pi, \xi') \mid (\gamma, \mu, \pi, \xi') \in \partial(p \cdot (f : p)^*, \xi)\}
\end{array}
$$

$$\partial(p \| q, \xi) = \{(\gamma_p\, \gamma_q, \mu_p \cup \mu_q, \pi_p \| \pi_q, \xi \bullet \xi' \bullet \xi'') \mid (\gamma_p, \mu_p, \pi_p, \xi \bullet \xi') \in \partial(p, \xi) \text{ and } (\gamma_q, \mu_q, \pi_q, \xi \bullet \xi'') \in \partial(q, \xi)\}$$

Example 4.1.9. Let p be the CFE defined in Example 4.1.6. In Example 4.1.6 we showed that the basic block p was equivalent to the CFE p_1. Thus, $\partial(p, \xi_0) = \partial(p_1, \xi_0)$, which can be computed as follows.

$$\begin{aligned} \partial(p, \xi_0) &= \partial(p_1, \xi_0) \\ &= \left\{ \begin{array}{l} (\mathrm{et}(p,1)\, x_{11}, o_1, p_2, \xi_0), \\ (\mathrm{et}(p,1)\, \overline{x_{11}}, 0, p_2, \xi_0), \\ (\overline{\mathrm{et}}(p,1), \epsilon, \epsilon, \xi_0) \end{array} \right\} \end{aligned}$$

The first quadruple represents when operation o_1 is scheduled in the first cycle. The second quadruple represents the condition that operation o_1 is not scheduled in the first cycle. The third quadruple represents the condition in which the basic block can exit at p_1, without scheduling any operations. □

Example 4.1.10. In this example, we show how storages and derived storages are used in the computation of derivatives. Let r be a 2-bit register whose possible values are $\{0, \ldots, 3\}$, and let us compute the derivatives for the CFE $p = ([r++])^\omega$. Then, the following are the possible derivatives for p.

$$\partial(p, \xi_0) = \left\{ \begin{array}{l} (true, 0, p, \xi_0 \bullet [r \leftarrow 1]), \\ (true, 0, p, \xi_0 \bullet [r \leftarrow 1] \bullet [r \leftarrow 2]), \\ (true, 0, p, \xi_0 \bullet [r \leftarrow 1] \bullet [r \leftarrow 2] \bullet [r \leftarrow 3]), \\ (true, 0, p, \xi_0) \end{array} \right\}$$

□

4.1.3 Control-Flow Expression Suffixes

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 multi-cycle simulation of the control-flow expression.

Definition 4.9 *Let p be a control-flow expression. $\partial^i(p,\xi)$ is defined recursively as follows:*

$$
\begin{aligned}
\partial^1(p,\xi) &= \partial(p,\xi) \\
\partial^i(p,\xi) &= \bigcup_{(\gamma,\mu,\pi,\xi') \in \cup_{j=1}^{i-1} \partial^j(p,\xi)} \partial(\pi,\xi')
\end{aligned}
$$

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

Definition 4.10 *Let p be an extended control-flow expression and let ξ be a storage. The pair (q,ξ') is a suffix of (p,ξ) if $q = p$ and $\xi = \xi'$ or if $\exists n,\gamma,\mu,\xi' \ : \ (\gamma,\mu,q,\xi') \in \partial^n(p,\xi)$.*

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

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

Although the formula presented above computes all the suffixes of a control-flow expression, the formula neither specifies that the number of suffixes is finite, nor 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 4.11 *Two control-flow expressions, p and q, are equivalent if one can be obtained from the other using the CFE axioms in Table 2 of Chapter 2.*

Example 4.1.11. The control-flow expression $(a \cdot b \cdot c)^\omega$ is equivalent to $a \cdot b \cdot c \cdot (a \cdot b \cdot c)^\omega$. $\square$

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

Proof: We are going to prove this theorem recursively on the number of CFE compositions without considering storages.

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

$$
\begin{array}{rcl}
|\cup_{i=0}^{\infty}\partial^i(p\cdot q,\xi)| & \leq & (|\cup_{i=0}^{\infty}\partial^i(p,\xi)| \times |\cup_{i=0}^{\infty}\partial^i(q,\xi)|) + \\
 & & |\cup_{i=0}^{\infty}\partial^i(\Delta(p) : q,\xi)| \\
 & \leq & N_p N_q + N_q \\
|\cup_{i=0}^{\infty}\partial^i(c_1 : p + c_2 : q,\xi)| & \leq & |\cup_{i=0}^{\infty}\partial^i(c_1 : p,\xi)| + |\cup_{i=0}^{\infty}\partial^i(c_2 : q,\xi)| \\
 & \leq & N_p + N_q \\
|\cup_{i=0}^{\infty}\partial^i(p\|q,\xi)| & \leq & |\cup_{i=0}^{\infty}\partial^i(p,\xi)| \times |\cup_{i=0}^{\infty}\partial^i(q,\xi)| \\
 & \leq & N_p N_q \\
|\cup_{i=0}^{\infty}\partial^i(p^{\omega},\xi)| & \leq & |\cup_{i=0}^{\infty}\partial^i(p\cdot p^{\omega},\xi)| \\
 & \leq & 2^{|\mathcal{C}|}|\cup_{i=0}^{\infty}\partial^i(p,\xi)| \\
 & \leq & 2^{|\mathcal{C}|}N_p \\
|\cup_{i=0}^{\infty}\partial^i((c : p)^*,\xi)| & \leq & |\cup_{i=0}^{\infty}\partial^i(c : p\cdot(c : p)^*,\xi)|
\end{array}
$$

$$
\begin{aligned}
& \leq 2^{|\mathcal{C}|}|\cup_{i=0}^{\infty}\partial^i(p,\xi)| \\
& \leq 2^{|\mathcal{C}|}N_p \\
|\cup_{i=0}^{\infty}\partial^i(\{r_1;\ldots r_n\},\xi)| & \leq |\cup_{i=0}^{\infty}\partial^i(p_1,\xi)| \\
& \leq N_{p_1} \\
|\cup_{i=0}^{\infty}\partial^i(\overline{\zeta}(q),\xi)| & \leq |\cup_{i=0}^{\infty}\partial^i(q,\xi)| \\
& \leq N_q
\end{aligned}
$$

In the computation of $|\cup_{i=0}^{\infty}\partial^i(p^\omega,\xi)|$, we used the fact that p and p^ω have the same number of suffixes. It is not hard to see that if (π,ζ') is a suffix of (p,ζ), then $(\pi\cdot p^\omega,\zeta')$ will be a suffix of (p^ω,ζ).

If we consider the number of different storages to be $|\mathcal{S}|$, then all expressions above are multiplied by $|\mathcal{S}|$, since each derivative can generate at most $|\mathcal{S}|$ storages. $\sharp$

Theorem 4.3 *For any control-flow expression p, there exists N such that for all $M > N$ $\cup_{i=0}^{N}\partial^i(p,\xi) = \cup_{i=0}^{M}\partial^i(p,\xi)$.*

Proof: Suppose $\cup_{i=1}^{N}\partial^i(p,\xi) = \cup_{i=1}^{N-1}\partial^i(p,\xi)$, for some N.

Note that $\cup_{i=0}^{N+1}\partial^i(p,\xi) = (\cup_{i=0}^{N}\partial^i(p,\xi))\cup(\partial^{N+1}(p,\xi))$, and $\partial^{N+1}(p,\xi) = \cup_{(\gamma,\mu,\pi,\xi')\in\cup_{i=1}^{N}\partial^i(p,\xi)}(\pi,\xi')$.

Since $\cup_{i=1}^{N}\partial^i(p,\xi) = \cup_{i=1}^{N-1}\partial^i(p,\xi)$, then $\partial^{N+1}(p,\xi) = \cup_{(\gamma,\mu,\pi,\xi')\in\cup_{i=1}^{N-1}\partial^i(p,\xi)}(\pi,\xi') = \partial^N(p,\xi)$, and $\cup_{i=0}^{N+1}\partial^i(p,\xi) = \cup_{i=0}^{N}\partial^i(p,\xi)$. $\sharp$

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.

4.1.4 Revisiting Exception Handling

In the previous sections, we used $\overline{\zeta}(q)$ as an alternative definition for the disable construct $\zeta(n,p)$. With the definition of a derivative, we can show that $\exists q$ such that $\zeta(n,p) = \overline{\zeta}(q)$ for a given $\zeta(n,p)$ in p.

Theorem 4.4 *$\exists q$ such that $\zeta(n,p) = \overline{\zeta}(q)$.*

Before we prove this theorem, let us extend rewriting to CFEs. Let p be a CFE and let p' be a sub-expression of p. Then, $\overline{\mathcal{R}}(p)[p' \leftarrow p'']$ substitutes the occurrence of p' by p'' in p.

> **Proof:** Let p' be a *n-uplink* of $\zeta(n,p)$ in p, action $a \notin S_p$, and let us consider $D = \{(\gamma, \mu, \pi, \xi) \in \partial^\infty(\overline{\mathcal{R}}(p)[\zeta(n,p) \leftarrow a], \xi_0) \mid \pi = \sum_{i=0}^{m} c_{i1} : a_{i1} \cdot p_{i1} \| \ldots \| c_{il} : a_{il} \cdot p_{il} \| c_\zeta : a \cdot q_\zeta\}$, where Σ denotes alternative composition.
>
> Let us assume p' is a sub-expression of q_ζ. Then, p' was the result of an expansion from a loop ($*$) or an unconditional repetition (ω). Thus, we can rewrite $a \cdot q_\zeta$ as $a \cdot q_1 \cdot p' \cdot q_2$, and what follows p' will be q_2. The CFE $c_\zeta : a \cdot q_\zeta$ becomes $c_\zeta : \overline{\zeta}(q_2)$.
>
> Let us assume now that p' is not a sub-expression of q_ζ. Then, $q_\zeta = q_1 \cdot q_2$, with $(a \cdot q_1 \xi)$ being a suffix of p' for some ξ, and (q_2, ξ') being a suffix of p, for some ξ', but not of p'. Thus, we can rewrite $c_\zeta : a \cdot q_\zeta$ as $c_\zeta : \overline{\zeta}(q_2)$. $\sharp$

4.2 Constructing the Finite State Representation

In the previous sections we were concerned with the computation of the suffixes of a control-flow expressions by derivatives. In this section, we present a procedure to obtain the finite state Mealy machine from a control-flow expression using derivatives. This machine will be called a *control-flow finite state machine*, or CFFSM for short, and it is the Mealy machine 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., $\lambda : Q \times 2^I \rightarrow 2^O$.

[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 2.3, 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.

This Mealy machine is related to the set of derivatives of p and to the storage ξ 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, ξ) of p, we associate a state $q_{s,\xi} \in Q$. In particular, q_0 corresponds to the state q_{p,ξ_0}, i.e., to the CFE p itself and the storage ξ_0 in which all registers have zero value.

The transition function (δ) and the output function (λ) are related to the CFE p in the following way. Let (s, ξ) be an irredundant suffix of a control-flow expression p, for which we are building the finite-state machine representation. Assume that $(\gamma, \mu, \pi, \xi') \in \partial(s, \xi)$. Thus, $\delta(q_{s,\xi}, \gamma) = q_{\pi,\xi'}$ and $\lambda(q_{s,\xi}, \gamma) = \mu$ in M.

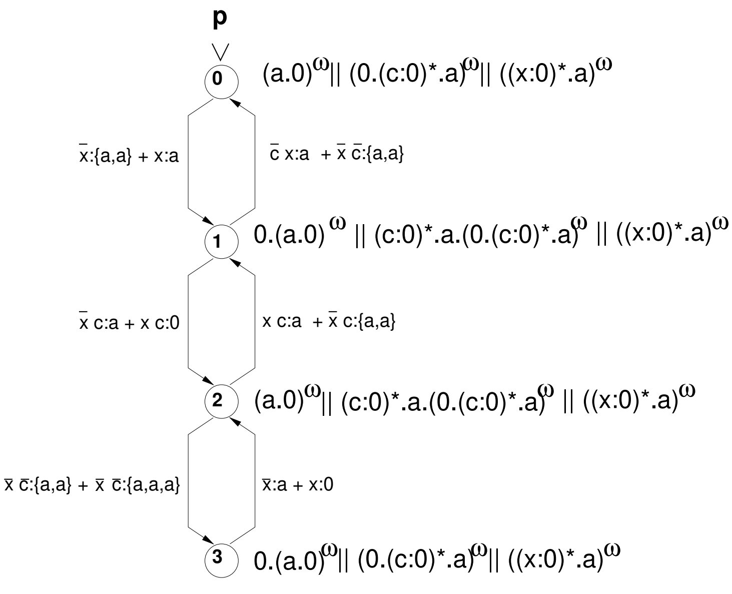


Figure 22: *Finite-state representation for synchronization synthesis problem*

Example 4.2.12. Figure 22 shows the finite-state representation for the synchronization example whose control-flow expression was presented in Example 3.2.9 ($p_1 || p_2 || p_3$). $\square$

Note that the derivative computation does not consider the synchronization constraints of *ALWAYS* and *NEVER* sets. We will handle these constraints by eliminating the transitions that invalidate the synchronization constraints. In the sequel, let $A_1, \ldots, A_n, A'_1, \ldots, A'_n$ be sets of actions, and let $\{A_1, \ldots, A_k\}$ be a multiset of actions grouped by the sets $A_1, \ldots, A_k$.

If a multiset of actions $\{A_1, \ldots, A_k\}$ belongs to the *ALWAYS* set and if at least one of the sets A_i intersects the actions specified on the transition, then at least one action of every group $A_j \in \{A_1, \ldots, A_k\}$ must be present in the transition. In addition to that, this transition should not intersect all the sets $A'_1, \ldots, A'_k$ at the same time if $\{A'_1, \ldots, A'_k\}$ is a multiset of actions of the *NEVER* set. This condition guarantees that the transition will not violate the synchronization requirements of the design.

Definition 4.12 *A transition $\delta(q, f)$ of a finite-state Mealy machine representation of a control-flow expression p is valid if the following conditions apply.*

- $\forall \{A_1, \ldots, A_k\} \in ALWAYS,\ \forall i \in \{1, \ldots, k\}$
  $$(\lambda(q, f) \cap A_i \neq \emptyset) \Rightarrow (\exists a_1 \in A_1, \ldots, a_k \in A_k \ \ \textit{such that} \ \ \{a_1, \ldots, a_k\} \in \lambda(q, f))$$
- $\forall \{A_1, \ldots, A_k\} \in NEVER,\ \forall i \in \{1, \ldots, k\}$
  $$(\lambda(q, f) \cap A_i \neq \emptyset) \Rightarrow \exists j \ \ \textit{such that} \ \ j \neq i \ \ \textit{and} \ \ (\lambda(q, f) \cap A_j = \emptyset)$$

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 4.1 *The initial state q_{p,ξ_0} 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 Figure 23 is used to compute the finite-state Mealy machine M of a specification. The algorithm works by traversing the finite-state machine

/* *Breath First Search of state space represented by CFE p* */
procedure *Construct_FSM*
{
  input: *cfe, storage, ALWAYS, NEVER*
  output: *finite-state machine M*
  fifo.init (cfe, storage) /* *initialize fifo with initial cfe* */
  while *(fifo* $\neq \emptyset$*)* {
    (cfe,storage) = fifo.first() /* *get cfe and storage on top of fifo* */
    mark((cfe,storage)) /* *mark cfe as traversed and make it a state* */
    derivative = ∂ *(cfe,storage)* /* *compute all cfe's one cycle apart* */
    $\forall(\gamma, \mu, \pi, \xi) : (\mathcal{G} \times \mathcal{M}^{\mathcal{A}} \times \mathcal{F} \times \mathcal{S}) \in$ *derivative* {
      /* *check if it violates ALWAYS and NEVER sets* */
      if (μ violates *ALWAYS*)
        continue
      if (μ violates *NEVER*)
        continue
      add edge ((*cfe,storage*)$, \gamma : \mu, (\pi, \xi)$) *to fsm*
      if *unmarked* $((\pi, \xi))$ /* *if suffix isn't a state, insert it in fifo* */
        fifo.insert (π, ξ)
    }
  }
  remove unreachable states
}

Figure 23: *Algorithm to construct finite-state representation*

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 4.2.13. If we apply the $NEVER = \{\{a\}, \{a\}\}$ constraint to the finite-state representation of $p_1||p_2||p_3$ (shown in Example 4.2.12), we obtain the finite-state representation of Figure 24 (b).

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

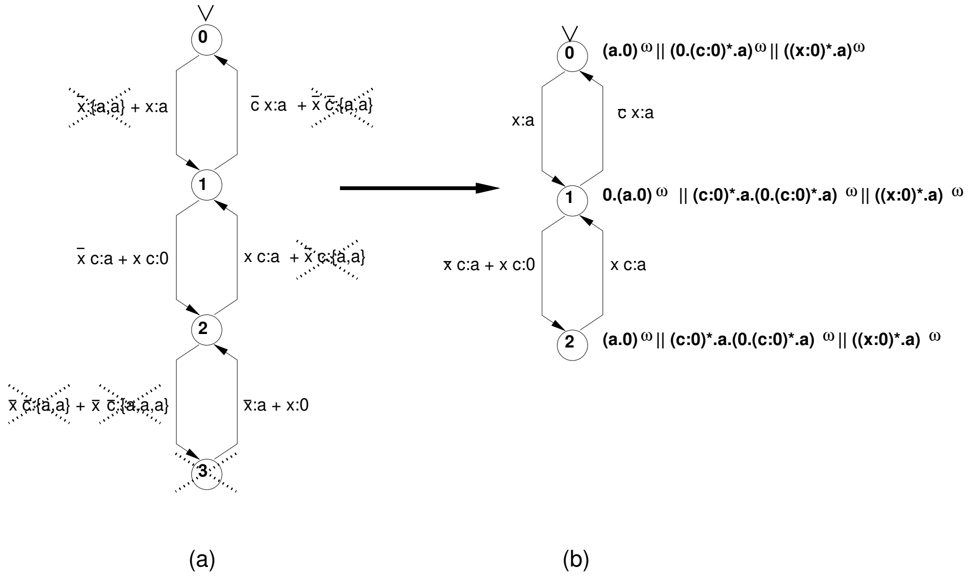


Figure 24: *Finite-state representation observing synchronization constraints*

4.2.1 Satisfiability of Design Constraints

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

Theorem 4.5 *Let p be a control-flow expression along with the synchronization constraints specified by the sets ALWAYS and NEVER. If the procedure Construct_FSM $(p, \xi_0, ALWAYS, NEVER)$ 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 Figure 23), 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. $\sharp$

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

4.3 Representation of CFFSM as a Transition Relation

In the previous sections we showed how we can build a Mealy automaton that satisfies the design constraints. Since the construction of the Mealy automaton involves the computation of a product machine, which can be computationally complex, we present in this section implementation issues for representing the Mealy automaton symbolically by a transition relation.

4.3.1 Characteristic Functions and Transition Relation

We revise in this section the concepts on representation of sets using characteristic functions and representation of finite-state machines using the transition relation, similarly to the definitions found in [TSL+90, CHJ+90, CM90, McM93, Hu95].

Let S be a set and $|S|$ its number of elements, and let $B = \{0, 1\}$. We define the bijective function $e : S \rightarrow B^m$ as an encoding from the elements of S to the m-dimensional Boolean space B^m. We say an encoding $e : S \rightarrow B^{|S|}$ is a one-hot encoding if exactly one of the bits of $B^{|S|}$ has value one. A m-hot encoding is the encoding $e : S \rightarrow B^n$ in which exactly m out of n bits have value one, and n is determined such that $|S| \leq C(n, m)$, i.e., the selection of m out of n. A binary encoding is the encoding $e : S \rightarrow B^{\lceil \log S \rceil}$.

Definition 4.13 *The characteristic function of a subset $S' \subseteq S$ is the function $\mathcal{X}_{S'}: B^m \rightarrow B$. For any element $s \in S$, $\mathcal{X}_{S'}(e(s)) = 1$ if $s \in S'$, and $\mathcal{X}_{S'}(e(s)) = 0$ if $s \notin S'$.*

Definition 4.14 *Let $f : B^n \times B^m \rightarrow B^n$ be a Boolean function. A transition relation $T : B^n \times B^m \times B^n \rightarrow B$ is defined as $\{(s, c, S) \in B^n \times B^m \times B^n \text{ such that } s = f(c, S)\}$.*

Equivalently, we can write

$$T(s, c, S) = \bigwedge_{1 \leq i \leq n} (s_i \equiv f_i(c, S))$$

4.3.2 Representation of the CFFSM Using the Transition Relation

The transition relation can be used for representing a CFFSM in the following way. Let $M = (Q, I, O, \delta, \lambda, q_0)$ be a Mealy finite-state machine, $\delta : Q \times 2^I \rightarrow Q$ and $\lambda : Q \times 2^I \rightarrow 2^O$, as defined in Section 4.2.

Let $e_q : Q \rightarrow B^n$ be a suitable encoding for the set of states Q, and let $e_i : 2^I \rightarrow B^m$ be a suitable encoding for the subsets of the set of inputs I. Let also s and S be vectors over encodings of states, i.e., $s = (s_1, \ldots, s_n)$ and $S = (S_1, \ldots, S_n)$ and let c be a vector over encodings on subsets of conditionals, i.e., $c = (c_1, \ldots, c_m)$. We denote next state and current state vectors by s and S, respectively.

We can interpret the current/next state vectors in circuit terms as follows. Each pair of state encoded variables (S_i, s_i) denote a D-flipflop in which the input is represented by variable s_i and the output is represented by variable S_i.

We referred to the encoding functions e_q and e_i as a suitable encoding for states and inputs. In the following discussion, we define the encoding functions for machine M, their relationship with the CFEs, and the representation for the output function λ.

Encoding of States

Since one of our objectives for representing the CFFSM by a transition relation is a more efficient representation of the CFFSM, we use different types of encodings to represent different parts of a CFE. In particular, we use binary encodings to represent the states in basic blocks, one-hot encodings to represent the states of sequential and alternative blocks of CFEs, and encodings consisting of different fields to model concurrency.

Basic Blocks: One of the key properties of a basic block is that it is a strictly sequential block with single entry point and possibly several exit points. Thus, we can efficiently encode a basic block by a binary representation of the cycles $\{1, \ldots, s\}$ of the basic block's possible schedules. We call this encoding function $e_{bb} : \{0, \ldots, s\} \rightarrow B^{\lceil s+1 \rceil}$. Note that we have to represent one more state than the number of cycles (which we call here cycle 0), since we have to represent a state in which no cycle of the basic block is executed.

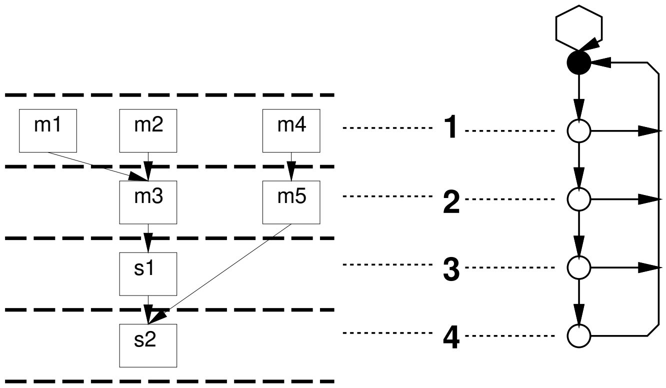


Figure 25: *Encoding for Basic Block of Differential Equation*

Example 4.3.14. In the basic block for the differential equation example, we represent its CFFSM by Figure 25. The following encoding is a possible binary encoding for the current states, with $S = (S_1, S_2, S_3)$.

$$e_{bb}(0) = \overline{S_1}\,\overline{S_2}\,\overline{S_3}$$

$$\begin{aligned} e_{bb}(1) &= S_1\,\overline{S_2}\,\overline{S_3} \\ e_{bb}(2) &= S_1\,S_2\,\overline{S_3} \\ e_{bb}(3) &= \overline{S_1}\,S_2\,\overline{S_3} \\ e_{bb}(4) &= \overline{S_1}\,S_2\,S_3 \end{aligned}$$

The encoding for the next states can be represented in a similar way. □

Sequential Blocks: In sequential blocks we use a one-hot encoding for the different states, i.e., $e_p : \mathcal{P} \rightarrow B^{|Q|}$, where $|Q|$ is the number of states in the sequential path of the control-flow expression.

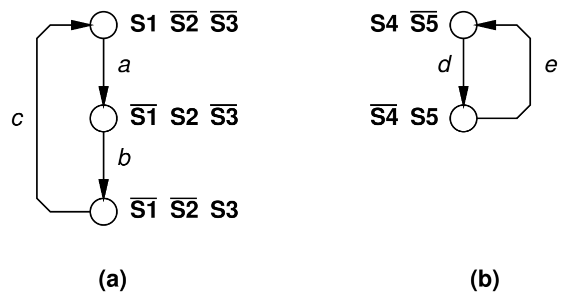


Figure 26: *Encoding for Sequential/Parallel Blocks*

Example 4.3.15. In the CFE $(a \cdot b \cdot c)^\omega$, we encode the suffixes $(a \cdot b \cdot c)^\omega$, $b \cdot c \cdot (a \cdot b \cdot c)^\omega$ and $c \cdot (a \cdot b \cdot c)^\omega$ by the Boolean functions representing the current states $S_1\,\overline{S_2}\,\overline{S_3}$, $\overline{S_1}\,S_2\,\overline{S_3}$ and $\overline{S_1}\,\overline{S_2}\,S_3$, respectively. The CFFSM corresponding to $(a \cdot b \cdot c)^\omega$ can be seen in Figure 26 (a). □

Parallel Blocks: One of the major sources for the computational complexity is the representation of concurrency in control-flow expressions. If we use a one-hot encoding for the parallel composition, then the number of variables representing the current and next state variables may be exponentially high. Thus, instead of explicitly computing the parallel composition, we compute the finite-state machines for each concurrent parts separately, and merge the finite-state machines of the concurrent parts. This corresponds to allowing a m-hot encoding of the parallel composition, if the m processes are composed in parallel, i.e.

$e_{\|} : \mathcal{P} \rightarrow B^m$. Alternatively, one can consider this encoding as a one-hot encoding for each sequential block composed in parallel.

Example 4.3.16. In the control-flow expression $p = (a \cdot b \cdot c)^\omega \| (d \cdot e)^\omega$, we first build the Mealy finite-state machines for $(a \cdot b \cdot c)^\omega$ and $(d \cdot e)^\omega$, as given by Figure 26.

Note that the encoding for the initial state of p is represented by the current state $S_1 \overline{S_2}\, \overline{S_3}\, S_4 \overline{S_5}$. □

Encoding of Inputs

In general, we do not know any relations among the inputs, so a conservative approach would be to generate a one-hot encoding for all conditionals and decision variables, i.e. $e_i : 2^I \rightarrow B^{|I|}$.

Recall that basic blocks define decision variables for each action which will be scheduled statically, i.e., for each a_i, and decision variables x_{ij}, exactly one of x_{ij} will be *true*, and the others will be *false* during synthesis. Thus, this corresponds to a one-hot encoding for these variables. We can reduce the number of variables necessary to represent the transition relation of a basic block if we replace the one-hot encoding over decision variables by a binary encoding. Suppose an action a_i can be scheduled at cycles $\{1, \ldots, s_i\}$, resulting in decision variables x_{ij}, $j \in \{1, \ldots, s_i\}$. Thus, an encoding for the decision variables of action a_i can be represented by function $e_{a_i} : \{1, \ldots, s_i\} \rightarrow B^{\lceil \log s_i \rceil}$, whose range is defined for Boolean variables $\underline{x}_{ij}$, for $j \in \{1, \ldots, \lceil \log s_i \rceil\}$. In a similar manner, we can define an encoding for the decision variables defined for each cycle.

Example 4.3.17. In the basic block $\{o_1 \rightarrow o_2, o_1 \rightarrow o_3\}$, let us assume that action o_1 can be scheduled in the first or second cycles. Thus, for o_1 we have decision variables x_{11} and x_{12}, that corresponds to when o_1 executes in the first or second cycles, respectively.

These decision variables can be encoded by the mapping $e_{o_1}(x_{11}) = \overline{\underline{x}_1}$ and $e_{o_1}(x_{12}) = \underline{x}_1$, where $\underline{x}_1$ is a Boolean variable created for action o_1. □

Representing the Output Function

Note that the output function for the finite-state Mealy machine is represented by the function $\lambda : Q \times 2^I \rightarrow 2^O$, i.e., for each transition, λ represents which outputs will be generated. We use here a modified representation for the λ function. Instead of representing the possible actions that can be generated for each transition, we represent which transitions generate a given output.

Definition 4.15 *Let $o \in O$ be an output of $M = (Q, I, O, \delta, \lambda, q_0)$. An action activation function $A_f : O \rightarrow B^n \times B^m \rightarrow B$ is a Boolean function defined for each output representing the transitions in which the output will be generated.*

We can write the action activation function for an output o in the following way. $A_f(o) = \{e_q(q)\ e_i(i)$ such that $q \in Q, i \in 2^I$ and $o \in \lambda(q, i)\}$.

Example 4.3.18. In the CFE of Example 4.3.15, we have the following action activation functions for actions a, b and c.

$$\begin{aligned} A_f(a) &= S_1\,\overline{S_2}\,\overline{S_3} \\ A_f(b) &= \overline{S_1}\,S_2\,\overline{S_3} \\ A_f(c) &= \overline{S_1}\,\overline{S_2}\,S_3 \end{aligned}$$

□

Exception Handling in the Transition Relation

Because we partitioned the computation of the finite-state machine at parallel blocks, we must take into account when the disable construct crosses the partitioned blocks.

Thus, for each sub-expression p_i of a CFE p, we define an input called *kill* which forces the CFE to quit its execution, i.e. the CFE p_i has an associated input $kill_i$ which guards the every transition of p_i's FSM by $\overline{kill_i}$. In a disable construct, if p_i is a n-uplink of $\zeta(n, p)$ in p, then executing this command forces the $kill_i$ input of p_i to be activated.

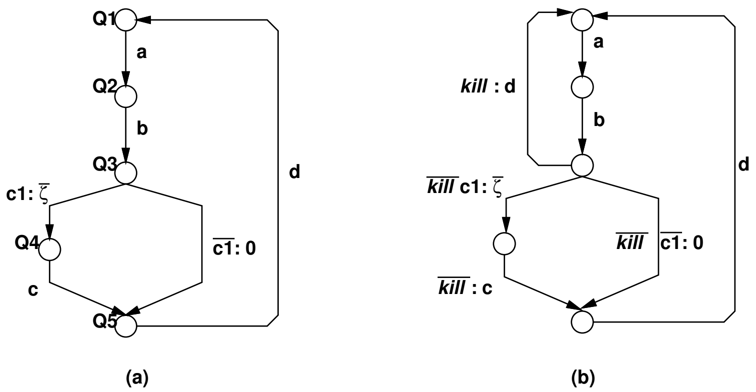


Figure 27: *Exception Handling in CFFSMs*

Example 4.3.19. In order to construct the CFFSM for the CFE $(a \cdot b \cdot (c_1 : \zeta(2,p) \cdot c + \overline{c_1} : 0) \cdot d)^\omega \equiv (a \cdot b \cdot (c_1 : \overline{\zeta}(d \cdot p) \cdot c + \overline{c_1} : 0) \cdot d)^\omega$, let us first consider the CFFSM for the CFE $(a \cdot b \cdot (c_1 : 0 \cdot c + \overline{c_1} : 0) \cdot d)^\omega$, which is represented in Figure 27 (a). We labeled the states of this figure as states Q_1,Q_2,Q_3,Q_4 and Q_5, as it can be seen in the figure. Since the alternative composition $(c_1 : \zeta(2,p) \cdot c + \overline{c_1} : 0)$ is aborted when ζ is executed, we create a signal *kill* that becomes *true* when ζ is executed.

Since we abort the execution of the CFFSM at state Q_3 if c_1 is *true*, signal *kill* will be assigned the truth value $e(Q_3)\, c_1$, where $e(Q_3)$ is the encoding for state Q_3. This signal will guard the execution of every path of the alternative composition until the CFE executes $d \cdot p$, as it can be seen in Figure 27 (b). Note that by setting *kill* to this value, state Q_4 is no longer reachable from the initial state. □

Putting it All Together: Computing the Transition Relation

The algorithm of Figure 28 computes the transition relation using the encoding algorithms presented in this section. We have to distinguish between three types of CFEs: basic blocks, parallel composition, and sequential composition for encoding purposes. *Collect_kill* collects all the transitions leading to the cfe coming from a disable construct. The final transition $tf_1 \times \ldots \times tf_n$ generates a synchronizer that waits until all transitions tf_i have occurred.

```
/* Computing Transition Relation of CFE */
procedure Compute_Tr
{
   input:  cfe,storage,kill
   output:  T,final transition,reset
   if (cfe is basic block) {
      return Compute_Tr_bb(cfe,kill)
   } else if (cfe = p_1||...||p_n) {        /* Compute Tr of each p_i separately */
      kill_|| = collect_kill(cfe) ∨ kill
      (T_i, tf_i, reset_i) = Compute_Tr(p_i, storage, kill_||)
      T = T_1 ... T_n
      reset = reset_1 ... reset_n
      tf = tf_1 × ... × tf_n
      return (T, tf, reset)
   } else if (cfe = ζ(n,p)) {          /* Make the leading transition to reset n-uplink block */
      Let p' be the n-uplink of ζ(n,p)
      collect_kill(p') = collect_kill(p') ∪ transition leading to ζ
   } else {
      kill_cfe = collect_kill(cfe) ∨ kill
      Compute (T, tf, reset) using derivatives
      return (T, tf, reset)
   }
}
```

Figure 28: *Algorithm to Compute Transition Relation of a CFE*

4.3.3 Computing Reachable States and Valid Transitions

We conclude this chapter by showing how we can use the transition relation to compute the set of valid transitions and reachable states.

The algorithm of Figure 29, which is similar to the algorithm presented in [TSL+90, CHJ+90, CM90], computes the set of reachable states in a CFFSM. In this algorithm, S_0 denotes the encoding for the initial state q_0 of M, and $\mathcal{X}_S = \mathcal{X}_s[(S_1,\ldots,S_n) \leftarrow (s_1,\ldots,s_n)]$ denotes a new characteristic function $\mathcal{X}_S$ in which every occurrence of s_i was replaced by S_i. This corresponds to the simulation of a clock tick if we consider the pair (S_i, s_i) to represent a D-flipflop.

In order to compute valid transitions, let us extend the action activation function A_f to a set of actions. We consider T to be the transition relation of a CFFSM in

```
/* Computing Reachable States in the CFFSM */
procedure Reachable_States
{
   input:  T, S_0
   output:  reachable
   reachable = S_0                              /* Initial states are reachable */
   S = S_0
   while (S ≠ ∅) {
      X_s = ∃S∃c(T(s,c,S)X_S                    /* Find next states satisfying T and X_S */
      X_S = X_s[(S_1,...,S_n) ← (s_1,...,s_n)]  /* Simulate a clock transition */
      S = S − reachable
      reachable = reachable ∪ S                 /* Collect next reachable states */
   }
}
```

Figure 29: *Algorithm to Compute Reachable States of a CFFSM*

the following definitions.

Definition 4.16 *Let* $A = \{a_1, \ldots, a_m\}$ *be a set of actions. Then, the action activation function over set* A *is defined as* $A_f(A) = \bigvee_{a_j \in A_i} A_f(a_j)$.

Theorem 4.6 *Let* $\{A_1, \ldots, A_k\} \in ALWAYS$*. A valid transition exists only in*

$$T\,(\bigwedge_{1 \leq i \leq k} A_f(A_i) | \bigwedge_{1 \leq i \leq k} \overline{A_f}(A_i))$$

.

Proof: This follows from the definition of $\{A_1, \ldots, A_k\} \in ALWAYS$ and from $A_f(A_i)$. ♯

Theorem 4.7 *Let* $\{A_1, \ldots, A_k\} \in NEVER$*. A valid transition exists only in*

$$T\,(\bigvee_{1 \leq i \leq k} \overline{A_f}(A_i))$$

.

Proof: This follows from the definition of $\{A_1, \ldots, A_k\} \in NEVER$ and from $A_f(A_i)$. ♯

4.4 Summary

In this chapter we showed how to represent a CFE as a Mealy finite-state machine by computing the derivatives of the CFE. Derivatives correspond to a cycle by cycle simulation of the CFE. We first showed how to compute the derivatives for the control-flow expressions defined in Section 2.2. Then, we presented a more general definition of derivatives for extended control-flow expressions that considered basic blocks, exception handling and register variables. Because of registers, the state of a control-flow expressions computed by a derivative had to consider the possible values of register variables, which were captured by stores and derived stores.

Analysis was performed at the finite-state machine level, where the unreachable states violating synchronization constraints were eliminated. We showed that if the resulting machine is empty, then the specification and its constraints cannot be satisfied.

For efficiency, we showed how to represent the finite-state machine by a transition relation, and we argued that an efficient encoding technique should be used in order to control the number of variables necessary to represent the transition relation.

Chapter 5

Synthesis of Control-Units

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

We will be considering two types of assignments for decision variables: the assignment of constant values (0 or 1) to decision variables, and the assignment of Boolean functions over states and conditionals. We will call the former assignment static scheduling and the latter dynamic scheduling. While static scheduling will be used to schedule operations in basic blocks (subject to constraints crossing basic blocks), dynamic scheduling will be used to synthesize the synchronization skeletons for more complex and loosely coupled interactions among the different parts of a concurrent description.

In the next section, we formulate the problem of synthesizing control-units from CFFSMs and we highlight the similarities between static scheduling and dynamic scheduling. Then, in Section 5.2, we review the techniques for static scheduling of operations in basic blocks subject to timing and resource constraints, and in Section 5.3, we show how we can extend these techniques to statically schedule operations in the basic blocks of a CFE. In Section 5.4, we show how to dynamically schedule operations for the loosely coupled parts of the design. Our algorithms will be cast as Integer Linear Programming (ILP) formulations, and solved by a Binary Decision Diagram (BDD) solver that we developed.

5.1 Obtaining Control-Units from the CFFSM

In this section, we formulate the problem of obtaining control-unit implementations from CFFSMs. This formulation will be applicable to both the static and dynamic scheduling techniques to be developed in the following sections of this chapter.

Figure 30 (a) presents the general methodology used for obtaining control-unit implementations from a CFE. From the CFE representing the specification and design constraints, in the previous chapter we obtained a CFFSM with the same behavior of the CFE.

Because we guarded design choices with decision variables, and in particular, because we guarded the execution time of actions by decision variables, the CFFSM contains the possible execution flows for the actions of the CFE. Thus, in order to obtain a feasible control-unit implementation for the concurrent parts of the CFE, we have to obtain an assignment for the decision variables of the CFE.

If we only consider static assignments to the decision variables, we obtain schedules for the actions in the CFFSM that do not change with the different states of the CFFSM. For example, the schedule of an operation relative to the beginning of a basic block will have a fixed number of cycles under all conditions of the system. As we mentioned before, these assignments are called static schedules.

We also consider the assignment of Boolean functions (over the transitions of the

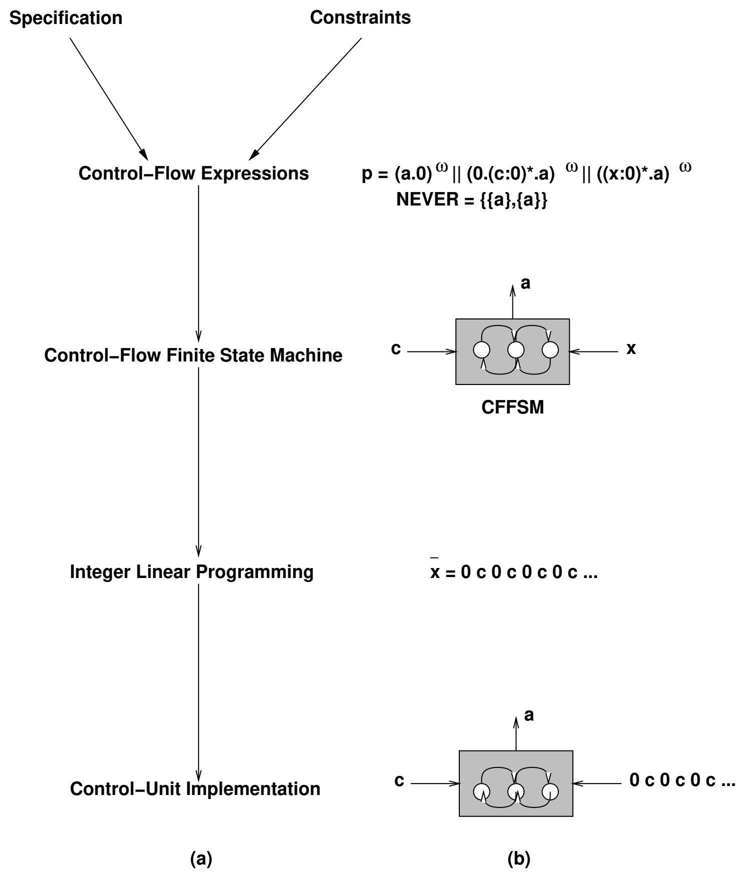


Figure 30: *Methodology for synthesizing control-units*

CFFSM) to the decision variables. This is used to satisfy more stringent constraints and constraints among the loosely coupled parts of the system. These assignments are called dynamic schedules.

Static scheduling and dynamic scheduling can be considered as restrictions of the behavior of the CFFSM. We consider in this chapter the CFFSM $M = (I, O, Q, \delta, \lambda, q_0)$, as defined in Section 4.2, where I is the set of inputs, O is the set of outputs, Q is the set of states, δ is the transition function ($2^I \times Q \rightarrow Q$), λ is the output function ($2^I \times Q \rightarrow 2^O$) and $q_0 \in Q$ is the initial state. A control-unit implementation will be the restriction of machine M to the set of assignments obtained for the decision variables. A formal definition of an implementation for M is given below.

Definition 5.1 *Let M be the CFFSM corresponding to some CFE p, and let X be a set of decision variables, i.e. $X \subseteq I$. 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.*
4. *The set of inputs of M' is $I - X$.*
5. *If every output $o \subseteq O$ is executed in some state of M, it will be executed in some state 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 $I' = I - X$, $Q' \subseteq Q$, $\delta' \subseteq \delta$, $\lambda' \subseteq \lambda$ and $\forall\ o \subseteq O, \lambda(o) \neq \emptyset \Rightarrow \lambda'(o) \neq \emptyset$. In addition to these requirements, M' must also satisfy additional constraints that will be imposed by the structure of the original specification. These additional constraints will vary depending whether we are obtaining static schedules for actions in basic blocks or dynamic schedules.

> **Example 5.1.1.** In Figure 30 (b), we present the steps used to obtain control-unit implementations from a CFE p. Starting from a CFE $p = (a \cdot 0)^\omega \| (0 \cdot (c : 0)^* \cdot a)^\omega \| ((x : 0)^* \cdot a)^\omega$, and a set of synchronization constraints, i.e. $NEVER = \{\{a\}, \{a\}\}$, we obtain a CFFSM corresponding to p. Then, we find a set of assignments to the decision variables of p over time, i.e. $X = \{x\}$ over time by casting the problem as an ILP. As mentioned previously in this section, the control-unit implementation will be the restriction of the CFFSM with respect to the assignments to the decision variable over time. □

In the next section, we will review static scheduling techniques for basic blocks cast as ILP instances. Then, in the following sections, we present static and dynamic scheduling techniques for the CFFSM.

5.2 Scheduling Operations in Basic Blocks

We review here the basic concepts for the scheduling problem that were originally defined in [HLH91, DGL92, Mic94] for basic blocks.

Let $O = \{o_1, \ldots, o_n\}$ be a set of operations. We assume that each operation o_i has an associated type $type(o_i) = k$, which indicates what kind of resource the operation uses.

Definition 5.2 *A scheduling problem is a mapping $\mathcal{T} : O \rightarrow \mathcal{N}^+$ that assigns with each operation a positive integer denoting the starting time of the operation, relative to the beginning of the basic block*

We denote the execution time for operation o_i by t_i, i.e., $\mathcal{T}(o_i) = t_i$. Let $\{r_1, \ldots, r_m\}$ be a set of precedence constraints. For any precedence constraint $o_{i_1} \xrightarrow{l} o_{i_2}$, the mapping $\mathcal{T}$ is such that the starting time of operation i_2 occurs after the starting time of operation i_1 by at least l cycles. Equivalently, $t_{i_2} - t_{i_1} \geq l$.

We are going to model the scheduling problem as an instance of Integer Linear Programming, which can be represented by following set of equations [Nem88].

$$
\begin{aligned}
\textbf{min} \qquad & \sum_i c_i x_i \\
& Ax = b \\
& x_i \in \{0, 1\}
\end{aligned}
$$

The solution to an ILP problem is an assignment to variables x_i such that they satisfy the set of constraints $Ax = b$, 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.

$$\sum_j x_{ij} = 1 \tag{5.1}$$

For every precedence constraint $o_{i_1} \xrightarrow{l} o_{i_2}$ in a basic block, the schedules of i_1 and i_2 are constrained by the equation $t_{i_2} - t_{i_1} \geq l$ 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.

$$\sum_j j\ x_{i_2 j} - \sum_j j\ x_{i_1 j} \geq l \tag{5.2}$$

Note that timing constraints between two operations in a basic block can be represented similarly. The precedence constraint $o_{i_1} \xrightarrow{l} o_{i_2}$ already represents the minimum time between o_{i_1} and o_{i_2} to be l. Maximum time constraints can be obtained by noting what happens when you multiply Inequality 5.2 by -1, which can be represented by the precedence constraint $o_{i_2} \xrightarrow{-l} o_{i_1}$ [HLH91, Mic94].

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.

$$\sum_{i \text{ such that } type(o_i)=k} x_{ij} \leq M_k \tag{5.3}$$

Example 5.2.2. In Figure 17 we presented the possible scheduling times for a piece of the differential equation solver. We represent below the Boolean variables defining the schedule for all operations of the basic block, 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.

$$\begin{aligned}
x_{m_1,1} &= 1 \\
x_{m_2,1} &= 1 \\
x_{m_3,2} &= 1 \\
x_{m_4,1} + x_{m_4,2} &= 1 \\
x_{m_5,2} + x_{m_5,3} &= 1 \\
x_{s_1,3} &= 1 \\
x_{s_2,4} &= 1 \\
2x_{m_3,2} - x_{m_1,1} &\geq 1 \\
2x_{m_3,2} - x_{m_2,1} &\geq 1 \\
3x_{s_1,3} - 2x_{m_3,2} &\geq 1 \\
4x_{s_2,4} - 3x_{s_1,3} &\geq 1 \\
(2x_{m_5,2} + 3x_{m_5,3}) - (1x_{m_4,1} + 2x_{m_4,2}) &\geq 1 \\
4x_{s_2,4} - (2x_{m_5,2} + 3x_{m_5,3}) &\geq 1
\end{aligned}$$

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

$$\begin{aligned}
x_{m_1,1} + x_{m_2,1} + x_{m_4,1} &\leq 2 \\
x_{m_3,2} + x_{m_4,2} + x_{m_5,2} &\leq 2 \\
x_{m_5,3} &\leq 2
\end{aligned}$$

□

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 5.3, but a variable that may take any integer value.

Example 5.2.3. For the set of equations presented in Example 5.2.2, 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 $\mathbf{min}\ 4x_{s_2,4}$.

If the objective goal is the minimization of resource cost, we replace the last 3 equations of Example 5.2.2 by the following equations.

$$
\begin{aligned}
x_{m_1,1} + x_{m_2,1} + x_{m_4,1} &\leq M_m \\
x_{m_3,2} + x_{m_4,2} + x_{m_5,2} &\leq M_m \\
x_{m_5,3} &\leq M_m
\end{aligned}
$$

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 $\mathbf{min}\ c_m M_m$. □

More recent advances in exact scheduling techniques for basic blocks can be found in [RB93, Geb91]. We will postpone any discussions until Section 5.5, when we compare these methods with the methods proposed in this chapter.

5.3 Static Scheduling Operations in CFFSMs

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, [TWL95] proposed a methodology of Behavior Finite State Machines (BFSMs) for representing sequential and conditional basic blocks (which are called behavioral states). In [YW], 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.

5.3.1 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, let T be its transition relation, and $\mathcal{D}_f$ be the set of basic blocks of p. For each basic block $d \in \mathcal{D}_f$, we assume the actions $A_d = \{a_1, \ldots, a_n\}$ are the actions defined in the precedence constraints of d, and $A' = \bigcup_{d \in \mathcal{D}_f} A_d$ is the set of actions defined in all basic blocks of p.

Recall that in the scheduling problem presented in Section 5.2, we defined three types of constraints for the Integer Linear Programming formulation. Equation 5.1 required that only one schedule for each operation was allowed. Inequality 5.2 defined the precedence constraints between two operations. Inequality 5.3 defined resource usage constraints inside a basic block.

In a CFE and its corresponding CFFSM, Equations 5.1 and 5.2 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 in Chapter 4. In that chapter, 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 5.1 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.

$$\sum_j e(x_{ij}) = 1 \tag{5.4}$$

Example 5.3.4. 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 $a_{i_1} \xrightarrow{l} a_{i_2}$ is a precedence constraint of a basic block, we can only allow the assignments to the corresponding decision variables of a_{i_1} and a_{i_2} (x_{i_1j} and x_{i_2j}, respectively) such that $t_{i_2} - t_{i_1} \geq l$, which can be represented in a form similar to Inequality 5.2.

$$\sum_j j\ e(x_{i_2j}) - \sum_j j\ e(x_{i_1j}) \quad \geq \quad l \tag{5.5}$$

Note that the function e in these set of ILP constraints acts as a linear transformation ($e(c_1f(x)+c_2g(x)) = c_1e(f(x))+c_2e(g(x))$) since it is a bijective function and it distributes over arithmetic addition and multiplication by constants. Consequently, Inequality 5.3 could be easily rewritten as:

$$\sum_{i\ such\ that\ type(o_i)=k} e(x_{ij}) \quad \leq \quad M_k \tag{5.6}$$

When the encoding function e is applied to the set of Inequalities 5.1, 5.2, 5.3, 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 chapter that BDDs can be used to efficiently solve these set of equations.

So far, we have shown how we could represent constraints that are limited to basic blocks. Recall that in Chapter 3 we defined statically satisfiable constraints, namely path-activated constraints, resource limiting constraints, and environment processes. These constraints impose additional restrictions to the decision variables, but they are not limited to basic blocks. We show in the sequel how to constraint the decision variables across basic blocks based on these of constraints.

Path-Activated Constraints

In Section 3.2.2, we defined a path-activated constraint as a convenience for representing timing constraints. Recall that 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 decision variables 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 $C_f : \mathcal{G} \rightarrow Q \times 2^I$ returns the set of transitions in which the Boolean formula guards the transition of the corresponding CFE.

Definition 5.3

$$\begin{aligned} A^*_f(l_i) &= \bigvee_{a_j \in l_i} x_j A_f(a_j) \quad \textit{if } l_i \textit{ is set of actions } \{\ldots, a_j, \ldots\} \\ &= A_f(a) \quad \textit{if } l_i \textit{ is action } a \\ &= gC_f(g) \quad \textit{if } l_i \textit{ is Boolean guard } g \end{aligned}$$

Note that in the case where $l_i = \{\ldots, a_j, \ldots\}$, 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 5.3.5. Let $p = (\{a_1 \rightarrow a_2, a_1 \rightarrow a_3, a_2 \rightarrow a_4, a_3 \rightarrow a_4\} \cdot (c : \{b_1 \rightarrow b_2, b_1 \rightarrow b_3, b_2 \rightarrow b_4, b_3 \rightarrow b_4\})^*)^\omega$, represented graphically in Figure 31 (a), and let both basic blocks to execute in at most 4 cycles. The CFFSM is presented in Figure 31 (b), where $\texttt{et}_{a4}$ corresponds to the condition when the first basic block requires 4 cycles to execute, and $\texttt{et}_{b4}$ 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

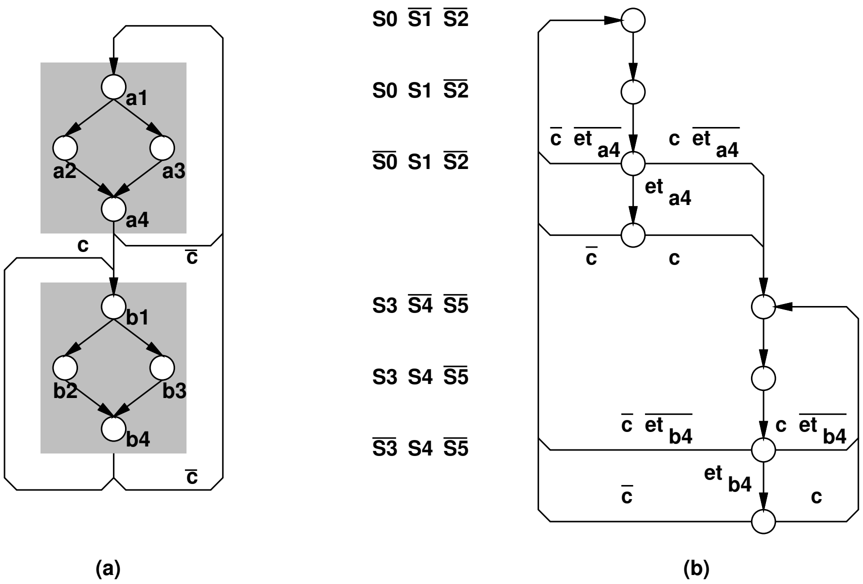


Figure 31: *(a) Graphical representation of CFE p and (b) CFFSM for p*

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.

$$\begin{aligned} \mathtt{et}_{a4} &= (e(x_{a_4 4}) \vee e(y_{a4})) \\ \mathtt{et}_{b4} &= (e(x_{b_4 4}) \vee e(y_{b4})) \end{aligned}$$

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

$$\begin{aligned} A^*_f(a_1) &= e(x_{a_1 1})(S_0\, \overline{S_1}\, \overline{S_2}) \vee e(x_{a_1 2})(S_0\, S_1\, \overline{S_2}) \\ A^*_f(c) &= c(\overline{S_0}\, S_1\, \overline{S_2}\, \overline{\mathtt{et}}_{a4} \vee \overline{S_0}\, S_1 S_2 \vee \overline{S_3}\, S_4\, \overline{S_5}\, \overline{\mathtt{et}}_{b4} \vee \overline{S_3}\, S_4 S_5) \end{aligned}$$

□

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 $A^*_f(l_3)$,

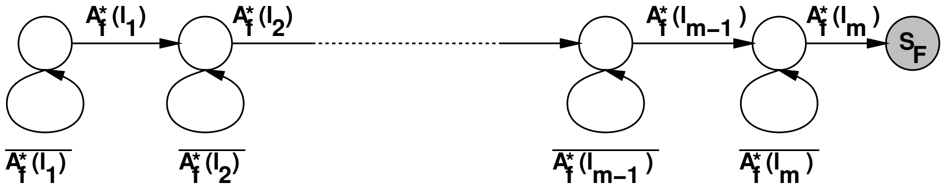


Figure 32: *Finite-State Machine Representing the Path-Activated Constraint*

and proceeding until we reach $A_f^*(l_m)$. Instead of existentially quantifying all inputs of the CFFSM during the traversal, as shown in the algorithm of Figure 29, 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 32. We traverse the CFFSM and the machine represented in Figure 32 until we reach state S_F. If the path constraint is $\mathbf{min}(n, [l_1, \ldots, l_m])$, 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 $\mathbf{max}(n, [l_1, \ldots, l_m])$, 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 $\mathbf{delay}(n, [l_1, \ldots, l_m])$, then we only keep the assignments to the decision variables in which S_F is reached during the traversal in exactly n cycles.

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

> **Proof:** First note that $A_f^*(a_1) = \bigvee_j e(x_{1j}) F_j^1(c, S)$, for some function $F_j^1(c, S)$. Thus, we can consider in this representation of $A_f^*(a_1)$ that $F_j^1(c, S)$ 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 $F_j^1(c, S)$ or not. Since $\sum_j e(x_{1j}) = 1$, then $\bigvee_j e(x_{1j}) F_j^1(c, S) = \sum_j e(x_{1j})$

$F_j^1(c, S)$, the same being valid for $A_f^*(a_2)$. Note also that the product of x_{1j_1} and x_{2j_2} can be replaced by the Boolean conjunction of x_{1j_1} and x_{2j_2}, since the only possible values for these variables are 0 and 1. Finally, because x_{1j_1} and x_{2j_2} can be represented $e(x_{1j_1})$ and $e(x_{2j_2})$ respectively, $x_{1j_1}x_{2j_2}$ can be replaced by the Boolean conjunction of $e(x_{1j_1})$ and $e(x_{2j_2})$.

The execution time for $t_{a_2} - t_{a_1} \geq n$ can be represented by the equation below, where *time* is a function returning the time when the action is executed.

$$
\begin{aligned}
\Leftrightarrow \quad & time(\sum_j e(x_{2j})F_j^2(c,S)) - time(\sum_j e(x_{1j})F_j^1(c,S)) \geq n \\
\Leftrightarrow \quad & \sum_j e(x_{2j})time(F_j^2(c,S)) - \sum_j e(x_{1j})time(F_j^1(c,S)) \geq n \\
\Leftrightarrow \quad & \sum_j e(x_{1j}) \sum_k e(x_{2k})time(F_k^2(c,S)) - \sum_k e(x_{2k}) \sum_j e(x_{1j}) \\
& time(F_j^1(c,S)) \geq n \\
\Leftrightarrow \quad & \sum_j \sum_k e(x_{1j})e(x_{2k})(time(F_k^2(c,S)) - time(F_j^1(c,S))) \geq n \\
\Leftrightarrow \quad & \sum_j \sum_k e(x_{1j})e(x_{2k})[time(F_k^2(c,S)) - time(F_j^1(c,S)) \geq n] = 1
\end{aligned}
$$

In the last equation, $[time(F_k^2(c,S)) - time(F_j^1(c,S)) \geq n]$ represents a Boolean function that returns 1 if we can traverse the CFFSM from $F_j^1(c,S)$ to $F_k^2(c,S)$ 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, $time(F_k^2(c,S)) - time(F_j^1(c,S))$ can be replaced by $k - j$.

$$
\begin{aligned}
\Leftrightarrow \quad & \sum_j \sum_k e(x_{1j})e(x_{2k})[k - j \geq n] = 1 \Rightarrow \\
\Leftrightarrow \quad & \sum_j je(x_{2j}) - \sum_j je(x_{1j}) \geq n
\end{aligned}
$$

♯

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 $x_{1j_1}, \ldots, x_{oj_o}$ 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 $\mathbf{min}(n, l_1 \ldots l_m) = \sum_{j_1}\sum_{j_2}\cdots\sum_{j_o}\bigwedge_i e(x_{ij_i})[Time(A_f^*(l_1), \ldots, A_f^*(l_m)) \geq n] = 1$*, where actions* $a_1, \ldots, a_o$ *are the actions whose decision variables* $x_{1j_1}, \ldots, x_{oj_o}$ *become constrained when traversing* $l_1, \ldots, l_m$ *for more than* $n-1$ *cycles.*

Corollary 5.2 $\mathbf{max}(n, l_1 \ldots l_m) = \sum_{j_1}\sum_{j_2}\cdots\sum_{j_o}\bigwedge_i e(x_{ij_i})[Time(A_f^*(l_1), \ldots, A_f^*(l_m)) \leq n] = 1$*, where actions* $a_1, \ldots, a_o$ *are the actions whose decision variables* $x_{1j_1}, \ldots, x_{oj_o}$ *become constrained when traversing* $l_1, \ldots, l_m$ *for less than* $n+1$ *cycles.*

Figure 33 presents the algorithm for minimum path-activated constraint and Figure 34 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 33 and 34, we traverse the finite-state machine presented in Figure 32 while traversing the transition relation T. For each state i of the finite-state machine of Figure 32, we keep the current transitions of the CFFSM leading to i in variable

```
/* Computing Minimum Path-Activated Constraint from CFFSM */
procedure Minimum_Path_Activated_Constraint
{
    input:  T, A*_f(l_1), ..., A*_f(l_m), n, Valid
    output:  constr
    T_1 = A*_f(l_1)
    for(i = 1; i <= m; i++) {
        T_{i+1} = T_i A*_f(l_{i+1})
        T_i = T_i ¬A*_f(l_{i+1}) Valid
    }
    /* traverse transition relation only n − 1 times */
    for(timer = n − 1; timer > 0; timer−−) {
        for(i = 1; i < m; i++) {
            /* Advance clock tick for T_i */
            T_i = ∃conditionals T_i
            F = T_i
            /* propagate F as far as possible */
            for(j = i; F ≠ 0 && j <= m; j++) {
                NT_j = NT_j ∨ F ¬A*_f(l_j) Valid
                F = F ¬A*_f(l_j) Valid
            }
            if (j == m + 1) constr = constr ∨ F
        }
    ∀i  NT_i = T_i
    }
    constr = ∀B ¬(∃S  constr)
}
```

Figure 33: *Algorithm to Compute a Minimum Path-Activated Constraint in a CFFSM*

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 32.

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

```
/* Computing Maximum Path-Activated Constraint from CFFSM */
procedure Maximum_Path_Activated_Constraint
{
   input:  T, A*_f(l_1), ..., A*_f(l_m), n, Valid
   output:  constr
   T_1 = A*_f(l_1)
   for(i = 1; i <= m; i++) {
      T_{i+1} = T_i A*_f(l_{i+1})
      T_i = T_i ¬A*_f(l_{i+1}) Valid
   }
   /* traverse transition relation only n times */
   for(timer = n - 1; timer >= 0; timer--) {
      for(i = 1; i < m; i++) {
         /* Advance clock tick for T_i */
         T_i = ∃conditionals T_i
         F = T_i
         /* propagate F as far as possible */
         for(j = i; F != 0 && j <= m; j++) {
            NT_j = NT_j ∨ F ¬A*_f(l_j) Valid
            F = F ¬A*_f(l_j) Valid
         }
         if (j == m + 1) constr = constr ∨ F
      }
   ∀i NT_i = T_i
   }
   constr = ∀B ∃S constr
}
```

Figure 34: *Algorithm to Compute a Maximum Path-Activated Constraint in a CFFSM*

that way, each computation of a path-activated constraint can be computed in a finite number of traversals ($O(m^2n)$). 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.

Example 5.3.6. A minimum execution time of 2 cycles between a_1 and c can be computed as follows. Let $e(x_{a_1 1}) = \overline{x_{a_1}}$ and $e(x_{a_1 2}) = x_{a_1}$. Thus, the

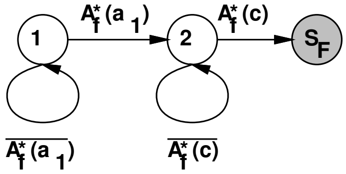


Figure 35: *Path-Activated Constraint FSM for* **min** $(2, [a_1, c])$

following activation functions were computed in Example 5.

$$\begin{aligned} A_f^*(a_1) &= \overline{x_{a1}}(S_0\,\overline{S_1}\,\overline{S_2}) \vee x_{a_1}(S_0\,S_1\,\overline{S_2}) \\ A_f^*(c) &= c(\overline{S_0}\,S_1\,\overline{S_2}\,\overline{\texttt{et}}_{a4} \vee \overline{S_0}\,S_1 S_2 \vee \overline{S_3}\,S_4\,\overline{S_5}\,\overline{\texttt{et}}_{b4} \vee \overline{S_3}\,S_4 S_5) \end{aligned}$$

As described in the algorithm of Figure 33, we traverse the machine of Figure 35, which is the machine describing the path-activated constraint **min** $(2, [a_1, c])$.

We begin the traversal of the transition relation from $A_f^*(a_1)$, i.e., from the transition $\overline{x_{a1}}(S_0\,\overline{S_1}\,\overline{S_2}) \vee x_{a_1}(S_0\,S_1\,\overline{S_2})$. Thus, considering $A_f^*(a_1)$ as the initial transition, the machine of Figure 35 steps to state 2. After the first transition of T, the set of reachable states of T becomes $\overline{x_{a1}}(S_0\,S_1\,\overline{S_2}) \vee x_{a_1}(\overline{S_0}\,S_1\,\overline{S_2})$. Since this new state intersects $A_f^*(c)$, when x_{a_1} is true, the machine of Figure 35 makes a transition from state 2 to state S_F when x_{a_1} is true.

No additional transitions are required for the path-activated constraint, and $x_{a_1}\overline{\texttt{et}}_{b4}\overline{S_0}S_1\overline{S_2}$ is the only transition that violates the path-activated constraint. After quantifying out the state and conditional variables of the constraint, we obtain the constraint $x_{a_1}\,\overline{\texttt{et}}_{b4} = x_{a_1}\,\overline{e(x_{a_4 4})}\,\overline{e(y_{a_4})}$, which is the only condition that violates the path-activated constraint. □

Resource Usage Constraints

We present in this section how we can generalize Inequality 5.6 to CFFSMs, in a manner similar to the extensions described for path-activated constraints. The basic idea is to use the transitions to detect any possibility of conflict in a set of actions, and then quantifying out all variables of the transition but the decision variables.

We use the resource limiting constraints that were defined in Section 3.2.2, i.e., the constraint **limit**$(n, \{a_1 \ldots a_m\})$, where n is the maximum number of concurrent executions of the actions in the set $\{a_1 \ldots a_m\}$.

In order to constrain the decision variables of actions in the set, we have to compute recursively function $C(n, a_1 \ldots a_m)$, which returns a Boolean function in which at least n out of m action activation functions $A_f(a_i)$ are *true*.

Definition 5.4

$$C(n, a_i \ldots a_m) = \begin{cases} 0 & \textit{if } n > m - i + 1 \\ 1 & \textit{if } n = 0 \\ (A_f(a_i) C(n-1, a_{i+1} \ldots a_m)) \ \vee \ C(n, a_{i+1} \ldots a_m) & \textit{otherwise} \end{cases}$$

For efficiency purposes, we implement $C(n, a_i \ldots a_m)$ using a dynamic programming paradigm [CLR90] such that we do not have to recompute $C(n, a_{i+1} \ldots a_m)$ every time it is needed.

We can now define **limit** by the following formula.

Definition 5.5 **limit**$(n, \{a_1 \ldots a_m\}) = \overline{\exists \ R \ \ C(n+1, a_1 \ldots a_m)}$.

where $R = S \cup s \cup$ *conditionals.*

This formula first computes the compositions of actions which yield more than the n available resources. Then, it existentially quantifies conditional variables, and present/next state variables of the action activation functions. Finally, the result is complemented, yielding the constraints on the decision variables such that less than n of the actions $\{a_1, \ldots, a_m\}$ are executed concurrently.

Example 5.3.7. In Example 5.2.2, let us assume the constraint **limit** $(1, \{a_1, a_2, a_3, a_4\})$ was specified. Let us also assume that $e(x_{a_i j}) = \overline{x_{a_i}}$ and $e(x_{a_i j+1}) = x_{a_i}$ for a_1, a_2, a_3 and a_4.

Thus, the following are the activation functions for a_1, a_2, a_3 and a_4.

$$\begin{array}{rcl} A_f^*(a_1) & = & \overline{x_{a_1}}(S_0\,\overline{S_1}\,\overline{S_2}) \vee x_{a_1}(S_0\,S_1\,\overline{S_2}) \\ A_f^*(a_2) & = & \overline{x_{a_2}}(S_0\,S_1\,\overline{S_2}) \vee x_{a_2}(\overline{S_0}\,S_1\,\overline{S_2}) \\ A_f^*(a_3) & = & \overline{x_{a_3}}(S_0\,S_1\,\overline{S_2}) \vee x_{a_3}(\overline{S_0}\,S_1\,\overline{S_2}) \\ A_f^*(a_4) & = & \overline{x_{a_4}}(\overline{S_0}\,S_1\,\overline{S_2}) \vee x_{a_4}(\overline{S_0}\,S_1\,S_2) \end{array}$$

According to the Equation presented before, **limit**$(1, \{a_1, a_2, a_3, a_4\})$ can be obtained by first computing $C(2, a_1 a_2 a_3 a_4)$, which is represented by the following equations.

$$
\begin{aligned}
C(2, a_1\, a_2\, a_3\, a_4) &= A_f^*(a_1) C(1, a_2\, a_3\, a_4) \vee C(2, a_2\, a_3\, a_4) \\
&\vdots \\
&= A_f^*(a_1)(A_f^*(a_2) \vee A_f^*(a_3) \vee A_f^*(a_4)) \vee \\
&\quad A_f^*(a_2)(A_f^*(a_3) \vee A_f^*(a_4)) \vee A_f^*(a_3) A_f^*(a_4) \\
&= (x_{a_1}(\overline{x_{a_2}} \vee \overline{x_{a_3}}) \vee \overline{x_{a_2}}\,\overline{x_{a_3}}) S_0\, S_1\, \overline{S_2} \vee \\
&\quad (x_{a_2}(x_{a_3} \vee \overline{x_{a_4}}) \vee x_{a_3} \overline{x_{a_4}}) \overline{S_0}\, S_1\, \overline{S_2}
\end{aligned}
$$

After quantifying out state variables and conditionals, we obtain the following equation:

$$
\exists S\ C(2, a_1\, a_2\, a_3\, a_4) = (x_{a_1}(\overline{x_{a_2}} \vee \overline{x_{a_3}}) \vee \overline{x_{a_2}}\,\overline{x_{a_3}}) \vee (x_{a_2}(x_{a_3} \vee \overline{x_{a_4}}) \vee x_{a_3} \overline{x_{a_4}})
$$

Thus, **limit** $(1, \{a_1, a_2, a_3, a_4\}) = \overline{x_{a_1}}(\overline{x_{a_2}}\, x_{a_3} \vee x_{a_2}\, \overline{x_{a_3}}) x_{a_4}$. □

Environment Processes

We show in this section the last type of constraint that is satisfied statically, namely environment processes. As defined in Section 3.2.2, an environment process is a process *env* which is a suitable representation of the environment and is composed in parallel with a portion of the control-flow expression for the specification p. Because $p \| env$ must constrain the assignments on decision variables statically, an environment process must be strongly coupled with the specification p by providing the necessary synchronization in terms of *ALWAYS* and *NEVER* sets.

In this section, we show how we can obtain constraints for the decision variables from these synchronization sets.

We showed in Section 4.3.3 that if the multiset $\{A_1, \ldots, A_k\}$ consisting of the sets of actions A_i were included in the *ALWAYS* set, then the set of valid transitions was contained in T $(\bigwedge_{1 \leq i \leq k} A_f(A_i) \vee \bigwedge_{1 \leq i \leq k} \overline{A_f}(A_i))$. Similarly, if the multiset

$\{A_1, \ldots, A_k\}$ consisting of the sets of actions A_i were included in the *NEVER* set, then the set of valid transitions was contained in $T\ (\bigvee_{1\leq i\leq k} \overline{A_f}(A_i))$.

In order to statically constrain decision variables, we have to extract these relations from $(\bigwedge_{1\leq i\leq k} A_f(A_i) \vee \bigwedge_{1\leq i\leq k} \overline{A_f}(A_i))$ and $(\bigvee_{1\leq i\leq k} \overline{A_f}(A_i))$ by keeping only the assignments to decision variables. For the *ALWAYS* case, we should consider the assignments for the decision variables in which the valid transitions contain either all the actions of the set executing at the same time or no actions of the set executing at the same time. For the *NEVER* case, we should consider only the valid transitions in which not all the actions in the set are executed at the same time.

Definition 5.6

$$\begin{aligned}
\texttt{always} &= \bigwedge_{\{A_1,\ldots,A_k\}\in ALWAYS} \exists\, R\ (\bigwedge_{1\leq i\leq k} A_f(A_i) \vee \bigwedge_{1\leq i\leq k} \overline{A_f}(A_i)\, Valid) \\
\texttt{never} &= \bigwedge_{\{A_1,\ldots,A_k\}\in NEVER} \exists\, R\ (Valid \bigvee_{1\leq i\leq k} \overline{A_f}(A_i))
\end{aligned}$$

where $R = S \cup s \cup conditionals$.

Example 5.3.8. In Example 5.3.5, we can also constrain the decision variables of p by composing p with an environment process. According to our assumptions about environment processes, recall that an environment process must be tightly coupled with p, for example the process $env = (a \cdot 0 \cdot 0 \cdot 0 \cdot (c : 0 \cdot 0 \cdot b)^*)^\omega$, with $ALWAYS = \{\{\{a\},\{a_1\}\},\{\{b\},\{b_4\}\}\}$.

This environment process not only constrains the delay between a_1 and c to be 2 cycles, but it also constrains the execution time for the second basic block to be 3 cycles. Thus, the constraint resulting from this environment process is $\texttt{et}_{a4}\ \texttt{et}_{b3}$. □

5.3.2 Exact Scheduling for Basic Blocks

We presented previously how to extend the constraint formulation for the decision variables to consider the specification represented by the CFFSM. In this section, we show how we can use Binary Decision Diagrams to represent the scheduling and resource usage constraints defined in the previous section. Are pointed out before, because these constraints are represented by mixed Boolean/arithmetic constraints, a conventional Integer Linear Programming solver will not be efficient for solving this set of constraints. Instead, we use a Binary Decision Diagram based solver similar to the ones described in [RB93, JS93]. However, instead of using a conventional Integer Linear Programming formulation, our solver can handle a more general formulation, as described below.

Let $X = \{x_1, \ldots, x_m\}$ be a set of Boolean variables, g_i and f_{ij} be Boolean functions over X and c_i, a_{ij} and b_j be numeric constants. A mixed Boolean/ILP instance can be represented by the following set of equations.

$$\begin{aligned} \mathbf{min} \qquad & \sum_i c_i g_i(X) \\ & \sum_i a_{ij} f_{ij}(X) \leq b_j \end{aligned}$$

Note that, in principle, this formulation can be converted into an Integer Linear Programming formulation by introducing more variables and equations. The original formulation of the scheduling problem which was represented by Inequalities 5.1,5.2 and 5.3, for example, contains more variables than the equivalent formulation represented by Inequalities 5.4,5.5 and 5.6.

In Appendix B we present a brief introduction to Binary Decision Diagrams, and we show how BDDs can be used to represent the Boolean/ILP constraints described above. The reader should refer to [BRB90, Bry86, Bry92] for a throughout explanation on BDDs, including details of an efficient implementation [BRB90].

Generating Objective Functions

In the general formulation for the Integer Linear Programming problem we presented in the previous section, we mentioned the objective function **min** $\sum_i c_i g_i(X)$, without mentioning how to obtain it. We present here a way to generalize the cost functions to range over decision variables of the CFFSM.

We consider here two type of objective functions, execution time and resource constraint. We represent objective functions by functional vectors, which are defined below.

Definition 5.7 *A functional vector is the vector $\underline{f} : \mathcal{B}^n \rightarrow \mathcal{B}^m$ in which $f_i(X)$ represents the i-th output bit of $\underline{f}$, where $X \subseteq \mathcal{B}^n$.*

Example 5.3.9. The functional vector

$$\underline{f} = \begin{bmatrix} a \vee b \\ ab \end{bmatrix}$$

corresponds to the Boolean functions $f_0 = a \vee b$ and $f_1 = ab$. □

A functional vector $\underline{f} = \{f_0, \ldots, f_m\}$ can be used to represent a mapping $\underline{f}^{\mathcal{N}} : \mathcal{B}^n \rightarrow \mathcal{N}$ by assuming that each f_i spans a partition of the Boolean space $\mathcal{B}^n$, i.e., $f_i f_j = \emptyset$ if $i \neq j$ and $\bigcup_i f_i = 1$. Moreover, for the assignments that satisfy the Boolean equation $f_i(X)$, the mapping $\underline{f}^{\mathcal{N}}(X)$ is assumed to have value i. We call function $\underline{f}^{\mathcal{N}}$ a numeric extension to the functional vector, and $\underline{f}$ the functional vector of $\underline{f}^{\mathcal{N}}$.

We use the notation $\underline{f}^{\mathcal{N}} = \sum_i i\ f_i = 0\ f_0 + 1\ f_1 + \cdots + m\ f_m$ to represent the numeric extension of the functional vector $\underline{f} = \{f_0, f_1, f_2, \ldots, f_m\}$ of length $m + 1$. We suppress the term $i\ f_i$ if $f_i = 0$.

Using this definition, we can apply arithmetic operations to functional vectors, such as addition and multiplication.

Definition 5.8 *Let $\underline{f}$ and $\underline{g}$ be two functional vectors of length m_1 and m_2, respectively, and let $\underline{f}^{\mathcal{N}}$ and $\underline{g}^{\mathcal{N}}$ be their numeric extension.*

In the addition of $\underline{f}^{\mathcal{N}}$ and $\underline{g}^{\mathcal{N}}$, represented by $\underline{h}^{\mathcal{N}} = \underline{f}^{\mathcal{N}} + \underline{g}^{\mathcal{N}}$, the functional vector for $\underline{h}^{\mathcal{N}}$ has size $m_1 + m_2$ and it satisfies the following relation.

$$h_k(X) = \bigvee_{i,j\,:\,i+j=k} f_i(X) g_j(X)$$

In the multiplication of $\underline{f}^{\mathcal{N}}$ and $\underline{g}^{\mathcal{N}}$, represented by $\underline{h}^{\mathcal{N}} = \underline{f}^{\mathcal{N}} * \underline{g}^{\mathcal{N}}$, the functional vector for $\underline{h}^{\mathcal{N}}$ has size $m_1 * m_2$ and it satisfies the following relation.

$$h_k(X) = \bigvee_{i,j\,:\,i*j=k} f_i(X) g_j(X)$$

Example 5.3.10. Let $\underline{f}^{\mathcal{N}}(x_1) = 3x_1 + 2\overline{x_1}$ and $\underline{g}^{\mathcal{N}}(x_1) = \overline{x_1} + 4x_1$, then:

$$\begin{aligned} \underline{f}^{\mathcal{N}}(x_1) + \underline{g}^{\mathcal{N}}(x_1) &= 3\overline{x_1} + 7x_1 \\ \underline{f}^{\mathcal{N}}(x_1) * \underline{g}^{\mathcal{N}}(x_1) &= 2\overline{x_1} + 12x_1 \end{aligned}$$

□

We can now define the objective function for the execution time of a basic block d.

Definition 5.9 $\underline{\mathtt{et}}^{\mathcal{N}}(d) = \sum_i i\ \mathtt{et}(d,i)$.

When composing the cost function for a whole path of the specification, we can compute the execution time for each basic block, and add the execution times using the definition above for the numeric extension of functional vectors.

Example 5.3.11. In Example 5.2.2, the numeric extension of the functional vectors denoting the execution time for the first and the second basic blocks are given below.

$$\begin{aligned} \underline{\mathtt{et}}^{\mathcal{N}}(1) &= 4\mathtt{et}_{a4} + 3\overline{\mathtt{et}}_{a4} \\ \underline{\mathtt{et}}^{\mathcal{N}}(2) &= 4\mathtt{et}_{b4} + 3\overline{\mathtt{et}}_{b4} \end{aligned}$$

The numeric extension of a functional vector denoting the execution time in the path consisting of the first and second basic blocks is given by $\underline{\mathtt{et}}^{\mathcal{N}}(1) + \underline{\mathtt{et}}^{\mathcal{N}}(2) = 8\mathtt{et}_{a4}\,\mathtt{et}_{b4} + 7(\overline{\mathtt{et}}_{a4}\,\mathtt{et}_{b4}|\mathtt{et}_{a4}\,\overline{\mathtt{et}}_{b4}) + 6\overline{\mathtt{et}}_{a4}\,\overline{\mathtt{et}}_{b4}$. □

We can also obtain a cost function for resources, using the constraints for resource constraints. Let $\{a_1, \ldots, a_n\}$ be a set of actions and let the cost of each resource be c. Recall that the function $\mathbf{limit}(m, \{a_1, \ldots, a_n\})$ computes the set of assignments for the decision variables such that at most m of the n actions execute at the same time. We use this function to determine how many actions will be executed concurrently in the definition presented below.

Definition 5.10 *The cost of using resources $\{a_1, \ldots, a_n\}$ is defined as*

$$\underline{resource}^{\mathcal{N}}(\{a_1, \ldots, a_n\}) = \sum_i (c\ i)\mathbf{limit}(i, \{a_1, \ldots, a_n\}) \bigwedge_{0 \le j < i} \overline{\mathbf{limit}}(j, \{a_1, \ldots, a_n\})$$

Example 5.3.12. Assuming the cost of each action a_i is 1, the cost function representing the number of concurrent uses of action a_i is shown below.

$$\begin{aligned}\underline{resource}^{\mathcal{N}}(\{a_1, a_2, a_3, a_4\}) \quad = \quad & 3(x_{a_1}\,\overline{x_{a_2}}\,\overline{x_{a_3}} \vee x_{a_2}\,x_{a_3}\,\overline{x_{a_4}}) + \\ & 2((x_{a_1} \vee \overline{x_{a_4}})(\overline{x_{a_2}}\,x_{a_3} \vee x_{a_2}\,\overline{x_{a_3}}) \vee \overline{x_{a_1}}\,\overline{x_{a_2}}\,\overline{x_{a_3}} \vee \\ & x_{a_2}\,x_{a_3}\,x_{a_4}) + 1\,(\overline{x_{a_1}}(\overline{x_{a_2}}\,x_{a_3} \vee x_{a_2}\,\overline{x_{a_3}})x_{a_4})\end{aligned}$$

□

Solving the Integer Linear Programming Problem

We present now an algorithm for solving the scheduling problem defined in the previous sections.

We split the solution method into two portions: *solve* and *commit*. *Solve* obtains a set of solutions that satisfies the constraints given by Inequalities 5.4,5.5 and 5.6, path-activated constraints, resource limiting constraints and environment processes, while minimizing the objective function given by the numeric extension of a functional vector.

Commit obtains a unique assignment to the Boolean variables from the set of satisfying assignments obtained by *solve*. The advantages of such methods are twofold. First, we can apply *solve* with respect to several objective functions before committing to a single solution. For example, the primary goal may be the minimization of execution time, and the secondary goal may be the minimization of resource usage. Second, because the complexity of the ILP (and the corresponding BDDs) will be dependent on the number of variables and in the number of constraints, it may be worth while to solve some basic blocks first in order to reduce the complexity of the ILP solution for the whole system, but instead of committing to a single solution for each part of the system being solved, we carry to the other basic blocks the set of satisfying assignments such that those parts will have more degrees of freedom when selecting an optimal solution.

Note that *solve* and *commit* steps are possible only because we are using a Boolean representation for the solution space of the ILP problem, since conventional ILP solvers will only give you a single solution.

The algorithm for *solve* is presented in Figure 36. We assume that *constr* is the conjunction of the constraints obtained previously, and $\underline{g}$ is a functional vector whose numeric extension represents a cost function.

procedure *Solve*
{
  input: *constr*, $\underline{g}$
  output: X /* *set of satisfying assignments* */
  foreach $(i = 0;\ i \leq |\overline{g}|;\ i++)$ {
    $X = g_i\ \ constr$
    if $(X \neq 0)$ **break**
  }
}

Figure 36: *Algorithm to Compute Solve*

When committing to a single solution, we have to restrict the behavior of the

CFFSM by projecting the assignments of the decision variables back to the CFFSM M in order to obtain an implementation M'.

Let X_0 be a set of assignments to decision variables which have been resolved by a *commit* operation. An implementation with respect to X_0 can be obtained by constraining the output function and the transition functions of M with respect to X_0. Since M is represented by the transition relation T and the action activation function $A_f(a)$ for all actions a of M, an implementation M' can be obtained by intersecting T and $A_f(a)$ with X_0, and then quantifying out the variables of X_0. Such operation has been defined by McMillan as an *AND-Exists* [McM93].

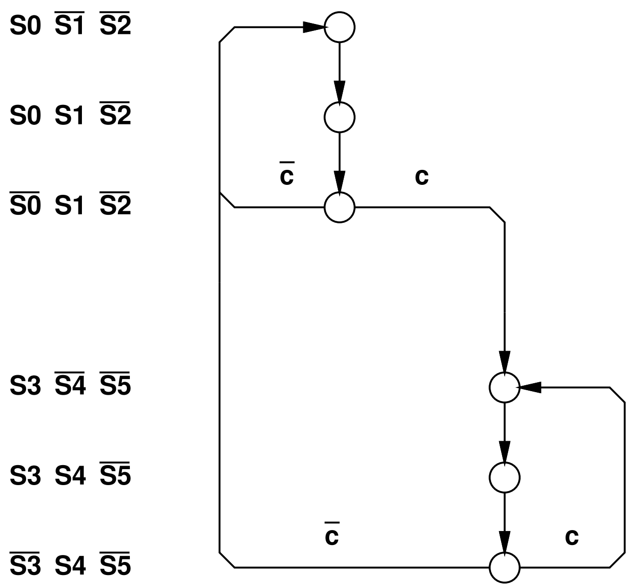


Figure 37: *Implementation for CFFSM*

Example 5.3.13. If we first solve the set of constraints using the cost function $\mathbf{min}\, \underline{\texttt{et}}^{\mathcal{N}}(a) + \underline{\texttt{et}}^{\mathcal{N}}(b)$, followed by solving the set of constraints using as a secondary goal the minimization on the number of resources a_i and b_i, we obtain the implementation of Figure 37.

In this implementation, actions a_1 and b_1 execute in the first transition of their respective basic blocks, actions a_2, a_3, b_3 and b_4 execute in the second transition of their respective basic blocks and actions a_4 and b_4 execute in the third transition of their respective basic blocks. $\square$

5.4 Dynamic Scheduling Operations in CFFSMs

Searching for static schedules for operations can lead to overconstrained problems if the concurrent parts are not coupled enough so that a static schedule can be found. For example, in the synchronization synthesis problem of Example 3.2.9, no static schedule can be found for the bus accesses of p_3 if p_1 and p_2 are not considered or the problem becomes overconstrained if p_1, p_2 and p_3 are considered and we attempt to find a static schedule that satisfies the synchronization constraints, because the accesses to the bus of p_2 are non-deterministic, i.e. they depend on the result of the evaluation of a conditional.

In this section, we will present a technique for generating control-unit implementations in which operations or basic blocks are synchronized with respect to processes that may not be tightly coupled. To determine when the operations can execute, we must consider the flow of control in conditional and concurrent paths to be independent of each other, as opposed to the procedure for finding static schedules presented in the previous section. In such systems, loops make the analysis of the control-flow to 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.

> **Example 5.4.14.** In Example 4.2.13, we represented the CFFSM for the system consisting of the concurrent processes $p_1 \| p_2 \| p_3$, where $p_1 = (a \cdot 0)^\omega$, $p_2 = (0 \cdot (c : 0)^* \cdot a)^\omega$ and $p_3 = ((x : 0)^* \cdot a)^\omega$, with the *NEVER* set being $\{\{a\}, \{a\}\}$.
>
> Recall that in this example, our goal here is the synthesis of a feasible control-unit for p_3 such that no two bus accesses occur at the same time. If we disregard the interactions of the other concurrent parts of the specification, then we will not be able to synthesize the control-unit that satisfies the design constraint.
>
> In Figure 24 (b), the CFFSM contains already all possible assignments to the

decision variables of the system. Note that there are two possible transitions from State 1, one to State 0 and another to State 2. The former transition occurs only if p_2 executes the bus access (represented by action a), while the latter transition only executes if p_2 does not make a bus access. Thus, any assignment to x will have to consider if p_2 is accessing the bus or not. $\Box$

We present in the sequel the ILP formulation for the synchronization synthesis problem. We consider $M = (I, O, Q, \delta, \lambda, q_0)$ to be the CFFSM representing the behavior of a CFE and $M' = (I, O, Q', \delta', \lambda', q_0)$ to be an implementation of M', as defined in Section 5.1.

The procedure we will present in this section can be considered as another extension to the scheduling technique presented before. In the previous section, because the design constraints were satisfied for all possible executions of the system, we did not have to consider the state of the CFFSM, but only decision variables. When we dynamically schedule the CFFSM, we will have to consider the global state of the system being synthesized in addition to the decision variables, since the decision variables will be functions of the transitions of the CFFSM.

In order to obtain an implementation 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'. Thus, 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 $f_{\mathbf{X}}$ and the second part $f_{\mathbf{C}}$. Now, for each state q_p, decision variable x of $f_{\mathbf{X}}$ and for each different Boolean formula $f_{\mathbf{C}}$ of q_p, we create a Boolean variable $x_{(q_p, f_{\mathbf{C}})}$. In the solution of the ILP problem, the variables $x_{(q_p, f_{\mathbf{C}})}$ are assigned 0-1 values such that if $f_{\mathbf{X}} \bigvee x \leftarrow x_{(q_p, f_{\mathbf{C}})} = 1$, then $\delta(q_p, f)$ belongs to M', i.e., if $f_{\mathbf{X}}$ evaluates to 1 when each variable x of $f_{\mathbf{X}}$ is assigned the value of $x_{(q_p, f_{\mathbf{C}})}$, then $\delta(q_p, f)$ will belong to M'.

Let us define also $f_{\mathbf{X}}^{\mathbf{C}}$ which stands for $(\forall x \in \mathbf{x}) f_{\mathbf{X}}(x = x_{p,f_{\mathbf{C}}})|_x$, i.e., the formula obtained by replacing every occurrence of $x \in f_{\mathbf{X}}$ by $x_{p,f_{\mathbf{C}}}$. We call a transition in which $f_{\mathbf{X}}^{\mathbf{C}} = 1$ a satisfying transition.

Finally, let $X = \{x_{p,f_{\mathbf{C}}}\} \cup \{y_p\}$ 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;

- If a state of M' has a satisfying transition to a state p', then state p' is also a state of M'. More formally, each state p' of M is a state of M' ($y_{p'} = 1$) if for every transition to p' ($\delta(f_{\mathbf{X}} f_{\mathbf{C}}, p) = p'$), $y_{p'} = \bigvee_p y_p f_{\mathbf{X}}^{\mathbf{C}}$.

- For each alternative composition in which the guards are decision variables, only one decision variable should be *true* for a given state of the finite-state machine.

  This statement is captured by following formula: $\sum x_{p,f_{\mathbf{C}}} = 1$, for all p and transitions $\delta(p, f) = p'$ and $\lambda(p, f) = a$ such that $x \in \mathbf{x}$.

- 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 1. The following equation captures this constraint:

$$\bigwedge_{\delta(p,f)=p' \wedge \lambda(p,f)=a} (\wedge x_{p,f_{\mathbf{C}}} \vee y_p) = 0$$

Example 5.4.15. Let us consider the finite-state machine of Figure 24 for the synchronization problem presented in Section 1.3.1. 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.

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

The first set of equations represent 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 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 $y_0 = y_1 = y_2 = 1$, $x_{0,1} = x_{1,\overline{c}} = x_{2,c} = 1$, $x_{1,c} = 0$. □

5.4.1 Selecting the 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 measures in order to be able to select the optimal implementation. Moreover, 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 CFEs, whose goals change with the different regions of the specification.

We can consider the representation of objective functions presented in this section as an expansion of the definitions of Section 5.3.2, when considering the framework of dynamic scheduling.

Selecting Minimum Scheduling Costs

A common optimization goal is synthesizing circuits whose running time is minimum. Since in basic blocks the synthesis of minimum schedules is equivalent to minimizing the execution time for the last operation of a basic block, every time an operation is delayed one cycle, we can insert an action 0 (corresponding to a delay of one cycle) before that operation. As a result, we can quantify the scheduling and synchronization constraints by counting the number of 0's inserted by an assignment to a decision variable in the CFFSM.

We can express the scheduling cost of an implementation by considering the synchronization and scheduling constraints of the specification. For synchronization 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 1, 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 $x^1, x^2, \ldots, x^n$ are decision variables. Every time x^i is assigned to 1, the latency of the process in which x^i was specified is increased by i.

Example 5.4.16. In the Example 5.3.15, we can represent the scheduling cost on x by the cost $\mathbf{min}\ (y_0 x_{(0,1)}) + (y_1(x_{(1,c)} \vee x_{(1,\overline{c})})) + (y_2 x_{(2,c)})$.

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

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 5.4.17. In Example 3.2.8, we rewrote the control-flow expression of the original specification in order to include binding constraints.

We can represent the binding cost of a implementation by the formula:

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

where $\beta(M_i)$ is the cost of component M_i, and $[\bigvee_{T_{M_i}}]$ 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'. □

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 CFEs, 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 5.4.18. In the specification of the ethernet coprocessor of Figure 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

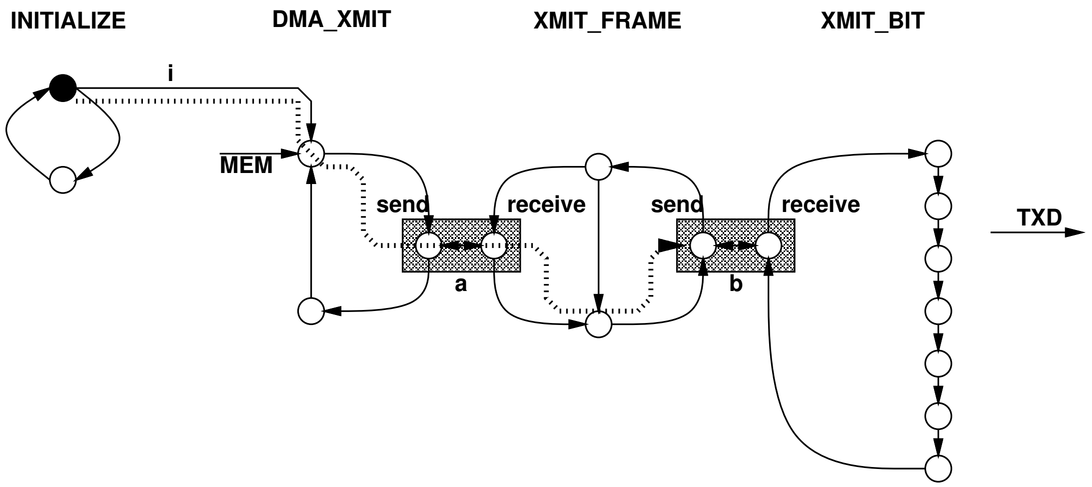


Figure 38: *Path cost selection in CFFSM*

be represented by action b, and the initialization of the transmission command by action i.

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. This path is represented in a dashed line in Figure 38, which represents the high-level view of the interactions among the processes *DMA_XMIT*, *XMIT_FRAME*, *XMIT_BIT* and *INITIALIZE*, which initializes the transmission of a frame. □

Note that the designer should be able to provide only cost measures by specifying which parts of the design he wants to tag 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.

5.4.2 Derivation of Control-Unit

We now show how we can obtain an implementation for the original processes from the finite-state machine M'. Note that M' not only represents the process that we want to synthesize, but also some of the environment constraints that were expressed by processes in CFEs. Thus, if we consider the original system to be represented by the CFE $p = p_1 \| \cdots \| p_n$, where p_i is the process for which a controller is sought, and processes $p_1, \ldots, p_{i-1}, p_{i+1}, \ldots, p_n$ are processes representing constraints and the environment, the final control-unit should be a restriction of M', when considering only the actions that are generated by p_i.

The problem then becomes deriving a machine $M_i = (I_i, O_i, Q', \delta_i, \lambda_i, q_0)$ for p_i from the implementation $M' = (I', O, Q', \delta', \lambda', q_0)$. In the new machine M_i, the set I_i of inputs to M_i correspond to I', which is already the set of conditional variables. O_i correspond 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.

Because machine M_i is obtained from machine M', which contains an assignment to the decision variables based on a global view of the system, the size of M_i will depend on two factors: the number of states of the CFFSM representing the system (M), and in the number of different dynamic schedules selected for M_i.

In practice, we would like to keep the number of possible schedules for a given operation small in order to keep the number of states of M_i small. 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 5.4.19. 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 5.3.15 eliminates the transition from state

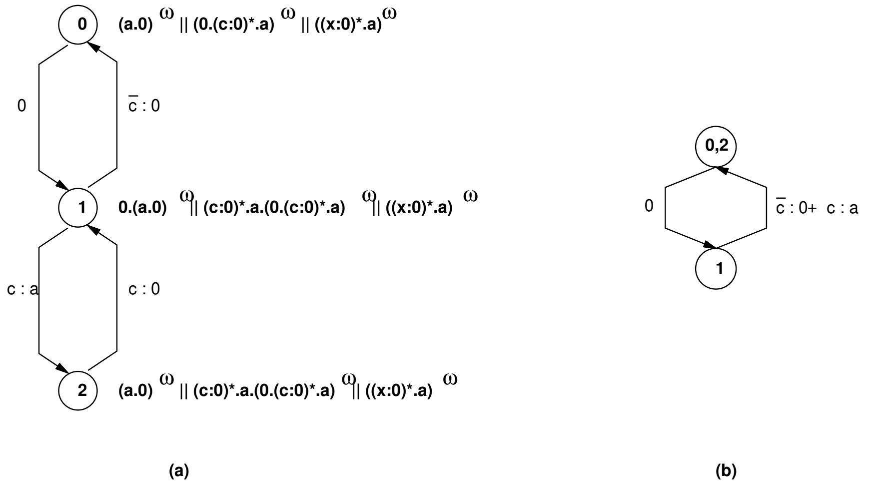


Figure 39: *Implementations for control-flow expression* $p_3 = ((x:0)^*.a)^\omega$

1 to state 2 when c is true. If we restrict the implementation on the actions generated by $p_3 = ((x:0)^*.a)^\omega$, we obtain the finite-state machine presented in Figure 39 (a). □

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 minimize the primary optimization goal (e.g. latency). We use the state minimizer *Stamina* [RHSJ94] to obtain the minimum number of states for the control-flow expression. In fact, in Example 5.4.19, an implementation with minimum number of states can be obtained with just 2 states (Figure 39 (b)).

Note that the number of states for the finite-state machine representing an implementation for a control-flow expression will have the number of states 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 concurrent CFEs is high, then the number of states of the finite-state machine will be much lower than the product of the number of states of the finite-state machine for each control-flow expression. 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 this subparts can be broken into smaller pieces and synthesized one at a time in order to reduce the number of states for the implementation.

5.5 Comparison with Other Scheduling Methods

We are going to analyze the procedures given previously to obtain an implementation that minimizes or maximizes the general cost functions.

Most previous approaches to scheduling and binding are usually restricted to single-source, single-exit control-data flow graphs [HP82, KLL91, HLH91, WT92, FKJM93, RB93, Mar90, Geb91], 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 DSPs [Lan91, NFCM92]. The major difference between these previous approaches to scheduling and the approach described in this chapter is that previous approaches viewed time as a linear order of events, i.e. a control-step i occurs always before control-step j if $i < j$, which is preserved in basic blocks. However, because we consider constraints crossing basic block boundaries, loops, synchronizations and concurrency, we cannot consider time a linear order any longer, and we have to resort to the branching structure of the CFFSM to represent the flow of time, which is captured by the states of the CFFSM.

In [FYDM93], a framework based on finite-automata was presented for modeling the design space of a high-level specification. Our work differs from this model in the following aspects. First, we handle several language constructs that are found in the specification of complex systems, such as exception handling and register variables. Second, we use a different encoding for basic blocks in order to reduce the number of states in the CFFSM. This compact representation does not exist in [FYDM93]. Third, we allow the user to specify flexible objective functions that can be used to capture represent higher-level design goals for optimization. Finally, we synchronize parts of the design for which no static schedule can be found.

The two methods that we presented have different complexities associated with them. Static scheduling can be used to schedule actions of basic blocks, subject to complex interface constraints. In these specifications, we assume that the system may consist of loosely coupled parts when applying design constraints, as long as the loosely coupled parts eventually synchronize. The running time of our algorithms for static scheduling should be exponential in the number decision variables. Since we used Boolean encoding in the decision variables for the actions in basic blocks, the complexity is $\mathcal{O}(2^{\sum \lceil \log n_a \rceil})$, where n_a is the number of control-steps defined for each action a. In addition to that, if we consider that not all basic blocks need to be scheduled at the same time, the complexity — although still exponential — can be brought down to reasonable running times for most of the problems.

The static scheduling technique can be used to schedule basic blocks, and it has the same complexity of [HLH91, RB93]. However, we may resort to dynamic scheduling for scheduling in some problems, such as the bus synchronization problem defined previously. In such cases, we allow systems to be loosely coupled and we can synthesize the concurrent parts running at different speeds or having complex interactions, for example, through synchronization. In [HP92, FKJM93], for example, synchronization among concurrent parts were considered, but only in a limited way. Dynamic scheduling also allows the implementation to reconfigure itself with respect to the system's state, since the controller may react differently to the conditionals of the system. In [GPR93], reconfiguration procedure for datapaths was described, but this reconfiguration is used only in case of failure. In our case, reconfiguration can be incorporated into the design as the environment.

Dynamic scheduling outperforms previous synthesis methods because it can handle loops, synchronization and complex interface constraints, and multi-rate execution of concurrent models. However, dynamic scheduling is computationally more expensive than these methods, since the number of variables that is solved is in the order of the number of transitions of the CFFSM in the worst case. However, not all operations need to be scheduled dynamically and most often, only basic blocks need to be synchronized. This can reduce the execution time for dynamic scheduling considerably. If we consider a specification with 10 basic blocks with 5 operations/basic block, although we have 50 operations to be scheduled in the specification, only 10 basic blocks need to be dynamically scheduled. In addition to that, if operations of a basic block do not need to be synchronized, then we do not need to dynamically schedule the basic block.

5.6 Summary

We presented in this chapter two complementary procedures for generating controllers from the CFFSM. The first procedure which has a lower execution time complexity, can statically schedule operations in basic blocks such that the final implementation satisfies path-activated constraints, resource limiting constraints and environment processes.

We showed that these constraints could be represented by traversals on the CFFSM if we keep the possible assignments for the decision variables during the traversal. This method allowed us to consider more general scheduling problems than the ones defined in the past, which were either restricted to basic blocks or restricted to sequential components alone.

The second procedure, with a higher execution time complexity, is used in the parts of the design that requires the generation of synchronization skeletons. We showed that if no static schedule can be obtained for the operations of a design, then we would have to resort to dynamic scheduling techniques, in which the schedules change over time with respect to the global view of the system.

Both of these methods benefited from using general cost functions that captured higher level objective functions, such as the minimization of the execution time over paths.

Chapter 6

Experimental Results

In this chapter, we present how real designs can be successfully synthesized using the techniques described in the previous chapters. We present control-dominated designs containing concurrent models, communication, and complex interface constraints to show how their control-units can be efficiently synthesized using a program we developed. This program and an embedded Binary Decision Diagram ILP solver were implemented in 25,000 lines of C code on a tool called *Thalia*, which is one of the parts of the *Parnassus Synthesis System*, shown in Figure 40.

In the next section, we show the effects of the encoding techniques described in Chapter 4 on the overall synthesis, and how such encodings affect performance. We will see that the use of such encodings is necessary in order to make Binary Decision Diagrams efficient to solve ILP problems.

Then, we show how this tool can be used to synthesize a protocol converter between a PCI protocol and a synchronous DRAM protocol, followed by the synthesis of two versions of the transmission block of the Ethernet controller, one using register variables to encode the different states of the protocol, and another one using exception handling mechanism. Finally, we will present the synthesis of a FIFO controller described in [YW]. Although this last specification is a sequential model, it

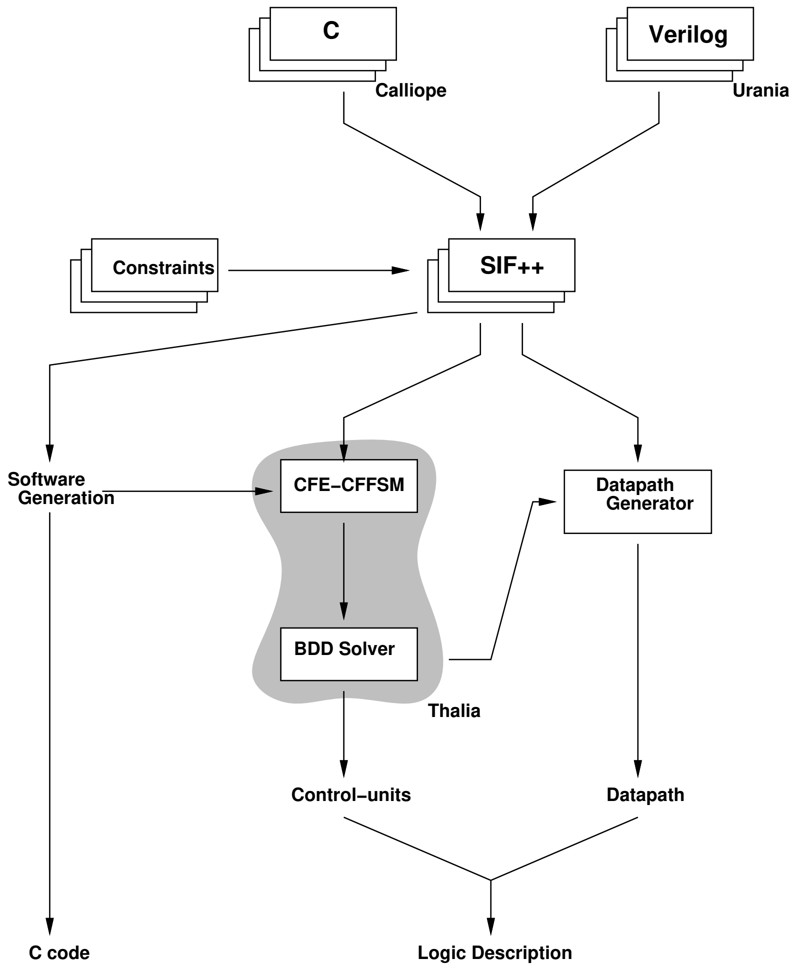


Figure 40: *Block diagram of Parnassus Synthesis System*

will enable us to address the limitations of the techniques presented in this thesis, when compared to existing tools, such as BFSM modeling and scheduling.

Unless otherwise stated, all execution times reported in this chapter will be for an Silicon Graphics INDY 4400 at 200 Mhz with 64 Mbytes of RAM.

6.1 The Effects of Encoding on the Synthesis Procedure

In this section, we will consider the effects of encoding in the overall synthesis technique. As pointed out in [DGL92], one of the main limitations of the 0-1 ILP formulation for scheduling is that as we increase the number of control-steps of the basic block, the number of variables for all operations increase. This can be seen in Table 3, where we used the *diffeq* basic block with a limit of one multiplier and one ALU performing addition, subtraction and comparison. In this table, the constraint size and the transition relation size were reported in terms of the number of BDD nodes. It is worth noting that there are 11 operations in the *diffeq* basic block and that no feasible solution exists for *diffeq* in less than 7 cycles with 1 multiplier and 1 ALU.

Control-steps	Constraint	Trans. Rel.	CPU Time (s)	Variables
7	172	85	0.36	52
8	2059	158	0.74	63
9	6016	209	1.40	74
10	14010	272	2.21	85
11	28490	344	4.34	96
12	52777	433	9.48	107
13	91228	536	19.89	118
14	149412	648	41.78	129
15	234298	772	81.88	140
16	354455	993	211.33	151

Table 3: *One-hot encoding for decision variables*

As it can be seen, the size of the ILP constraints represented by BDDs soon becomes unmanageable due to the increase on the number of variables. The encoding used for the decision variables in the scheduling problem of *diffeq* was a one-hot encoding, and, as we mentioned in Chapter 4, we could use an encoding using fewer variables by encoding the decision variables using Gray or a binary encoding. In Tables 4 and 5, it can be seen that we obtain much better results with respect to the compactness of the constraint representation and execution time if we encode the decision variables using a Gray or binary encodings. In fact, it can be seen that Gray encoding performs a little better in terms of compactness on the sizes of the constraints, and this encoding for the input variables was suggested in Chapter 4.

Control-steps	Constraint	Trans. Rel.	CPU Time (s)	Variables
7	83	94	0.46	26
8	1044	202	0.52	33
9	2580	219	0.76	33
10	6162	342	1.15	37
11	11137	335	1.69	37
12	22376	563	3.78	44
13	35892	562	6.22	44
14	54664	702	10.22	44
15	79760	651	14.89	44
16	113486	921	25.50	44

Table 4: *Binary encoding for decision variables*

6.2 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 [PCI95] 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. Figure 41 shows the diagram of 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 SDRAMs burst mode are limited to at most 8 transfers on the same row that are one

Control-steps	Constraint	Trans. Rel.	CPU Time (s)	Variables
7	83	93	0.22	26
8	1026	180	0.49	33
9	2586	210	0.73	33
10	6054	296	1.24	37
11	11101	313	1.64	37
12	21491	469	3.34	44
13	34822	490	5.94	44
14	52751	582	9.23	44
15	78338	583	16.43	44
16	109407	789	22.41	44

Table 5: *Gray encoding for decision variables*

cycle apart from each other.

The PCI bus has a notion of a master or initiator of a request and a slave or request target. The master (in this case the microprocessor) begins a bus cycle by asserting the signal `FRAME#` [1], writing the desired address on `AD[31::0]` lines and setting the bus command to the lines `C/BE#`. The master also asserts the signal `IRDY#` to indicate when it is ready to transmit or accept data. The target (in this case the memory), upon receiving a request, asserts `DEVSEL#`. When the target is ready to transmit or accept data, it asserts `TRDY#` signal.

A PCI bus cycle begins with a address phase, followed by one or more data phases. Figure 42 shows examples of PCI bus write cycles and Figure 43 shows the PCI bus read cycles, both taken from [PCI95]. Every data phase completes when `IRDY#` and `TRDY#` are asserted low by the master and the target, respectively.

Wait states can be inserted in the data phase by either the microprocessor or by the memory. The microprocessor inserts wait states during a transaction by de-asserting the signal `IRDY#` during a data phase, and the memory inserts wait states

[1] We are using the convention used by the PCI specification manual in which a signal followed by `#` indicates that the signal is asserted low

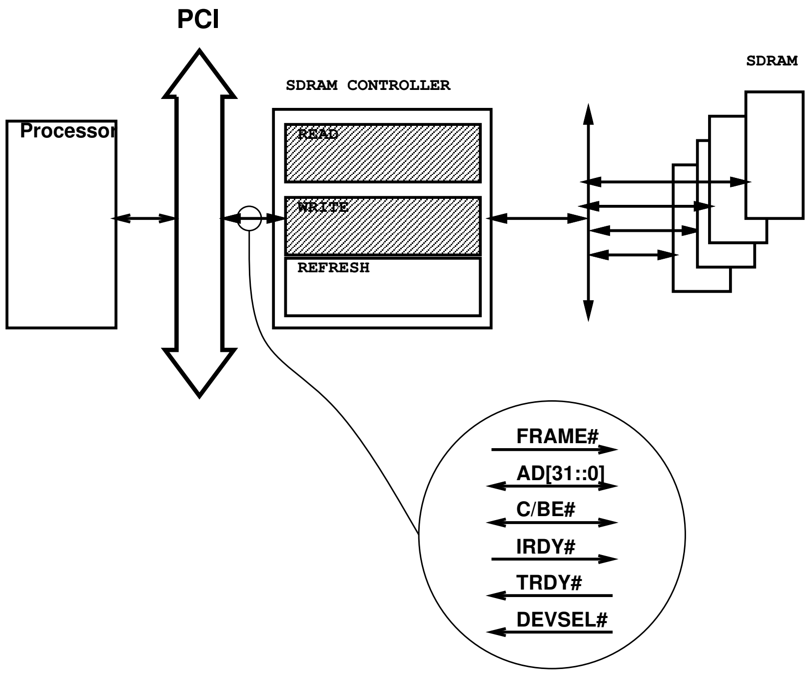


Figure 41: *Protocol conversion for PCI bus computer*

during a transaction by de-asserting the signal `TRDY#` during a data phase.

The synchronous DRAM [NEC93] protocol begins by a row address selection (`RAS`) phase followed by a column address selection (`CAS`) phase. If the previous `CAS` phase used the same row address as the row address that will be initiated, then the `RAS` phase can be omitted, thus saving a clock cycle. We present an example of a multiple `CAS` phases sharing the same `RAS` phase in Figure 44.

A `CAS` access can be either a read access or a write access, which is determined by signal $\overline{\texttt{WE}}$. The first `CAS` phase of Figure 44 is a read cycle and the second `CAS` phase is a write cycle. Associated with each `CAS` phase, the SDRAM has a programmable burst length. SDRAMs have a burst length of 1,2,4, 8 bytes or a full page. In the example of Figure 44, we assumed a burst length of 2, which is represented by accesses `a` and `b`. The latency of a reading cycle can also be programmable. In the SDRAM [NEC93], reading latency can be programmable to 1,2 or 3 cycles. In the example shown in

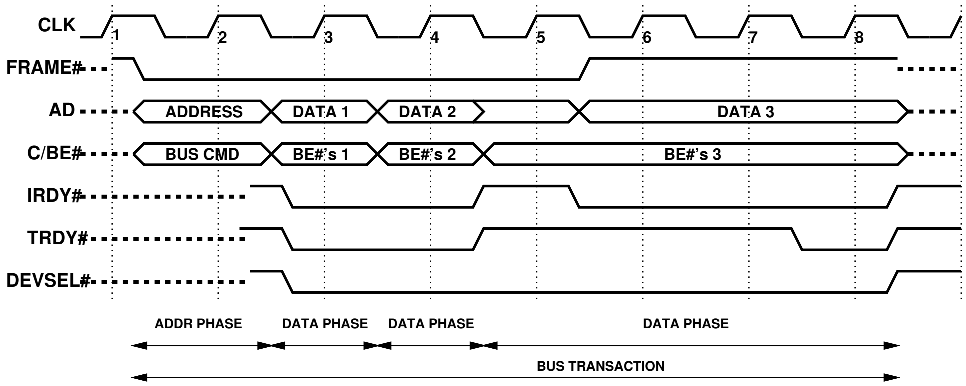


Figure 42: *PCI write cycle*

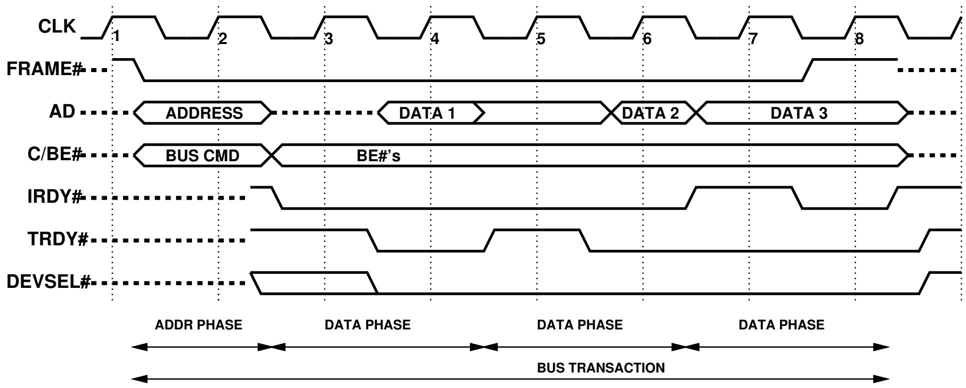


Figure 43: *PCI read cycle*

Figure 44, we assumed a latency of 1 cycle.

We implemented the four models for the reading and writing cycles of the PCI local bus and the SDRAM in 230 lines of a high-level subset of Verilog HDL, with the corresponding CFEs having similar complexity. These models are predefined libraries that can synchronize with any circuit. We thus use the technique of dynamically scheduling the transfers of each specification in order to synchronize the transfers between PCI and the SDRAM. The combined machine obtained from a PCI and SDRAM models is then used to synthesize a unique controller that is smaller than the two separate controllers.

Table 6 shows the number of states for the controllers in terms of a Mealy machine,

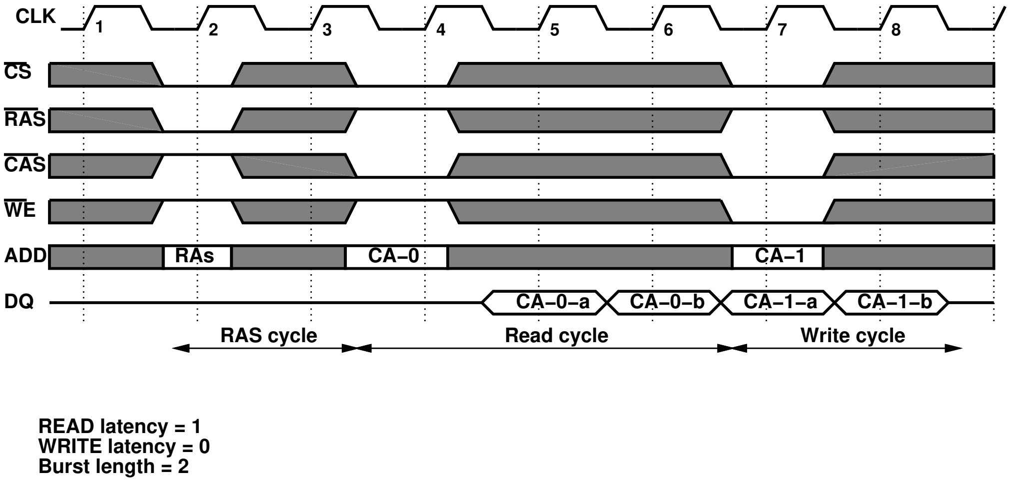


Figure 44: *SDRAM read and write cycles*

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.

Model	States PCI	States SDRAM	States PCI/SDRAM	Execution Time	Actions	Conds.	Dec. Vars.
READ	7	15	34	3.5 s	16	8	6
WRITE	6	7	30	1.6 s	15	8	3

Table 6: *PCI/SDRAM protocol conversion example*

6.3 Control-Unit for *Xmit_frame*

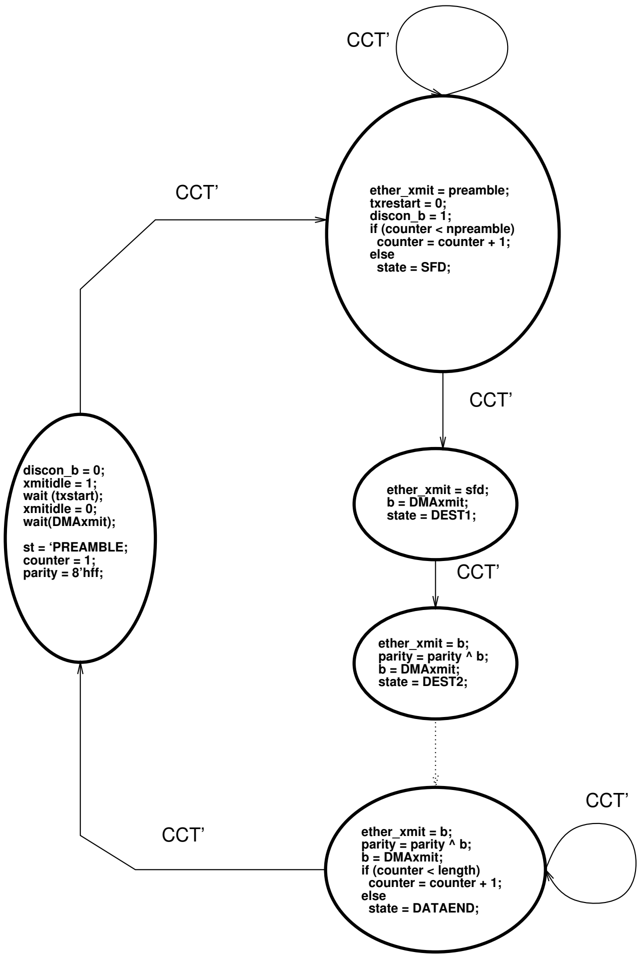


Figure 45: *Program state machine for process xmit_frame*

We consider here the Ethernet coprocessor of Figure 2. In that figure, let us focus on the transmission unit. As mentioned in Example 5.4.18, the transmission unit is composed of 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 constrains the rate at 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 Figure 45 [HLS], and it was also specified with an exception handling mechanism, i.e., the *disable* command of Verilog HDL. In the former implementation, because we have to abort the transmission of a frame if CCT becomes true, we implemented the program state machine with a *while* loop which pools signal CCT, and a *case* statement on variable *state*, which determines the next state of the program state machine to be executed. Note that this state variable is not part of dataflow and it should be incorporated into the control-unit for *xmit_frame*.

Since the execution of the *while* loop should be aborted if CCT becomes true, we re-implemented the specification for the program state machine of process *xmit_frame*. In this new specification, we execute a sequential code that traverse the different program states of Figure 45, and we execute a watchdog in parallel with the new sequential code. If condition CCT becomes true, then the watchdog will disable the execution of the concurrent block containing both the sequential code and the watchdog. A graphical representation of the specification can be seen in Figure 46.

Table 7 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 size of the transition relation in terms of BDD nodes. The fifth 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 specifications, we were able to obtain two comparable implementations with the same number of states.

In the table of Figure 7, note the difference between the sizes of the transition relation of both implementations. Although the complexity of the CFE in the program state machine case is larger than the complexity of the specification using the disable

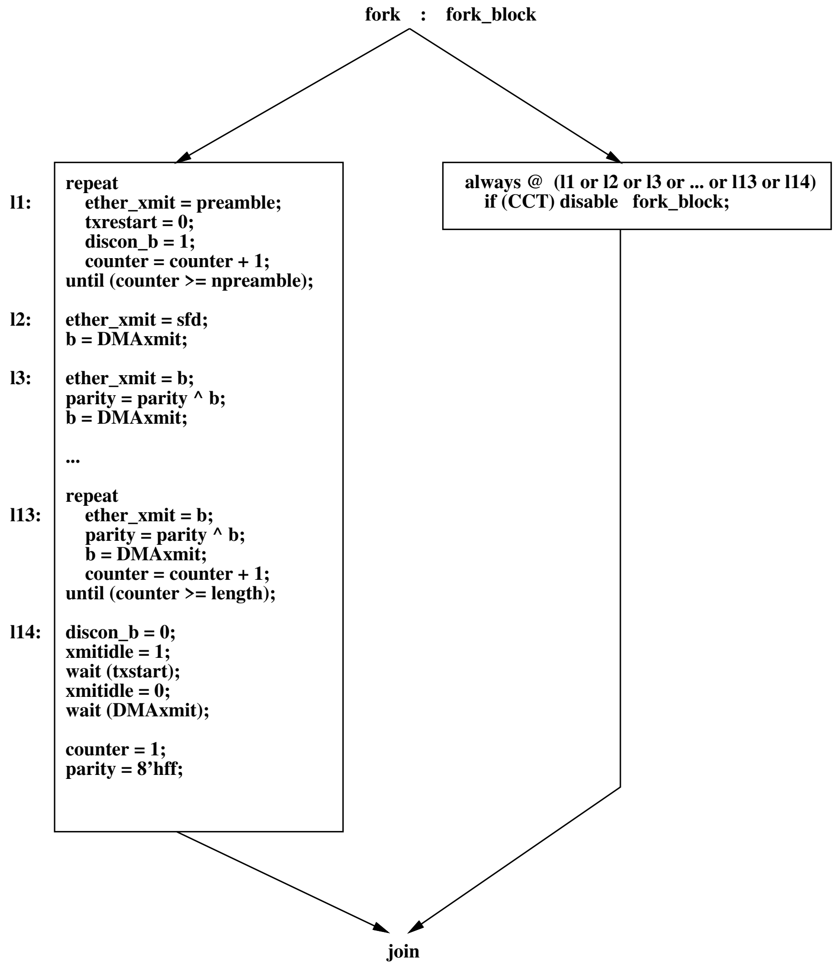


Figure 46: *Implementation of program state machine with exception handling*

construct, it would still not account for this large difference. Another reason for this discrepancy is due to the variable ordering chosen for the BDD variables. Binary Decision Diagrams are very sensitive to variable ordering and a bad choice for variable ordering can result in exponentially large BDDs. When computing the transition relation, we placed the conditionals and register variables on the top, and we grouped the Boolean variables belonging to basic blocks together.

The reader should recall that the program state machine implementation of *xmit_frame* has a state variable that was incorporated into the control-unit of

	# States		Constraint	Trans. Rel.	Time
xmit-frame (except.)	178	90	327	3022	4.3s
xmit-frame	178	90	995	24439	32.21s

Table 7: *Results for the synthesis of xmit_frame*

xmit_frame. This variable interacts with all basic blocks representing the states of the program state machine, as it can be seen in Figure 45. As a result, no good variable ordering can be found for this variable with respect to the ordering of variables created for each basic block.

In order to smooth out the effects of a bad variable ordering for the state variable, we ran both specifications on our program with the BDD using dynamic variable ordering [Rud93]. The results are reported in Table 8.

	# States		Constraint	Trans. Rel.	Time
xmit-frame (except.)	178	90	327	3022	4.35s
xmit-frame	178	90	899	14149	402.64s

Table 8: *Results for the synthesis of xmit_frame with dynamic variable ordering of BDDs*

6.4 FIFO Controller

In this section, we will compare the results of our approach with the specification of a FIFO controller that was presented in [YW]. For this example, which is a sequential model, we will show that we will not be able to obtain as good results as the ones reported by [YW]. We will explain the design choices that we made that led to these results. The reader must remember, however, that the approach presented in [YW] cannot be used to synthesize the controllers for the examples presented previously.

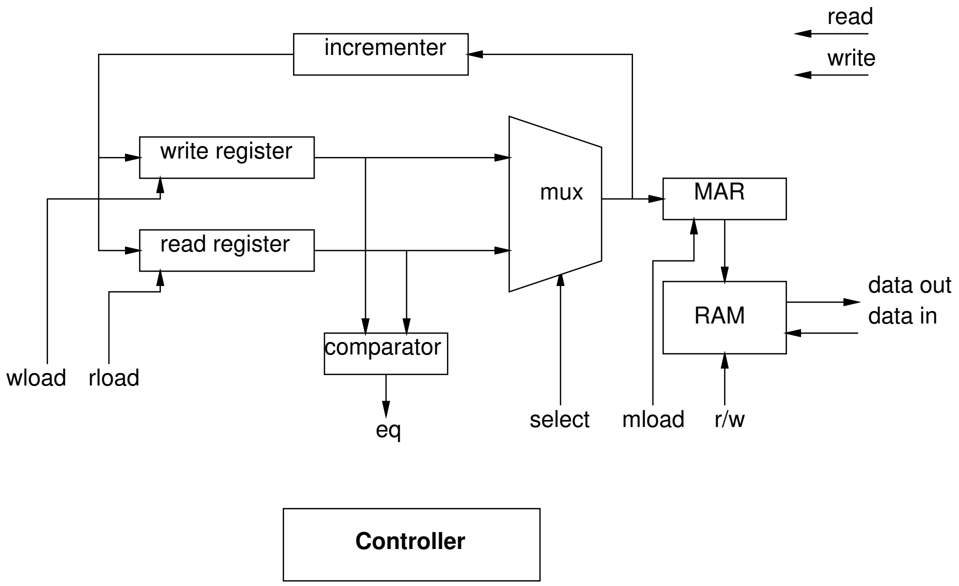


Figure 47: *Datapath for FIFO controller*

Figure 47 presents the datapath for a FIFO controller. In this datapath, `wload`, `rload`, `select`, `mload` and `R/W` are signals that must be generated to sensitize the paths of the datapath. Signals `eq`, `read` and `write` are input signals to the controller.

A high-level view of the system is presented in Figure 48. In this finite state machine, also called in [YW] a BFSM, the states corresponds to basic blocks of the specification. For example, the `IDLE` basic block is represented by the CFE $\{mux(1) \xrightarrow{0} rload(1); mux(1) \xrightarrow{0} mar(1); mar(0) \xrightarrow{0} r/w(1); mar(1) \xrightarrow{1} mar(0); rload(1) \xrightarrow{1} rload(0)\}$, where $signal(i)$ represents the condition of setting $signal$ to value i. In addition to the timing constraints implicit in the basic blocks, there are a number of path based constraints in the specification that must be satisfied as well. For example, one of the path based constraints that must be satisfied is that at least two cycles must occur between two writes of value 0 to signal r/w, i.e. $\mathbf{min}(2, r/w(0)\ r/w(0))$. We are able to obtain a controller for this specification with 20 states before optimization in 1.778 s, with a constraint size of 86 BDD nodes and a transition relation of 2113 BDD nodes.

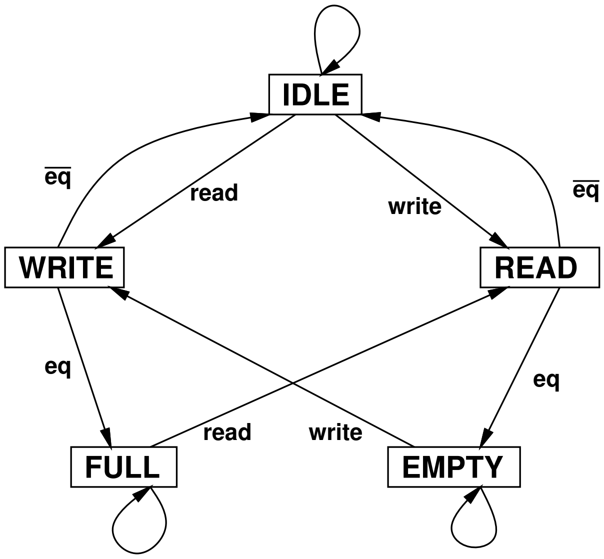


Figure 48: *High-level view of FIFO controller*

In the solution found in [YW], the `IDLE-READ-IDLE` loop is reported to have 3 cycles. Using our approach, we obtained the same loop with 5 cycles. The reason for this difference can be seen if we analyze how we handle constraints. We assumed the 0-1 ILP model for basic blocks, in which each basic block will take at least one cycle to execute when all operations of the basic block can be executed in parallel. Now, let us consider what happens when a path exists in the specification that traverses n basic blocks. In our model, we know that these n basic blocks will execute at least in n cycles.

This restriction does not exist in the BFSM algorithm proposed in [YW]. In this algorithm, the constraint on the minimum execution time is transferred from the basic blocks to the loops. From Chapter 2 and Chapter 4, we know that all loops must execute in at least one cycle. By removing the 1 cycle delay constraint of basic blocks, the algorithm of [YW] is able to better compress the execution time of sequential paths, and as a result, it is able to generate faster implementations such as the one shown in this section. We will address this issue and some additional improvements for our technique in the next chapter.

We must emphasize, however, that our approach is more complete in the sense it can consider constraints traversing concurrent models, which cannot be considered by [YW]. We can also consider register variables to be part of the specification, which cannot be easily incorporated by the BFSM representation.

Chapter 7

Conclusions and Future Work

7.1 Summary

We considered in this thesis modeling, analysis and synthesis techniques for concurrent and communicating, control-flow dominated designs, called *system-level designs*. For these specifications, current synthesis tools cannot handle concurrency, synchronization and exception handling and so they often achieve suboptimal results.

In order to best capture the degrees of freedom available in system-level designs, 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. In particular, we focused in the following aspects of system-level designs.

> *Modeling.* We presented an algebraic model called *control-flow expressions* to represent the control-flow of a system-level design. This model allowed us to capture most of the control-flow constructs of specification languages for hardware, such as Verilog HDL, VHDL, StateCharts, Esterel and the C programming language. Because it is common practice in structured programming

to use variables to represent parts of the control-flow behavior of a specification, we allowed some variables of the specification to be incorporated into control-flow expressions. This added greater flexibility to our model. We also allowed control-flow structures to break the normal flow of execution, which was captured by an operation similar to the *disable* statement of Verilog.

Constraints. We showed how to specify complex design constraints of a system. These constraints were not limited to timing and resource constraints, but considered also the synchronization of concurrent parts of the design. In addition, we were able to specify and consider constraints crossing concurrent blocks of the design, which are generally ignored in other synthesis tools.

Analysis and Synthesis. We showed that control-flow expressions can be analyzed by building the corresponding finite-state machine, called a control-flow finite-state machine (CFFSM), which encodes the possible design choices, and incorporates design constraints. We presented two synthesis techniques for scheduling operations that satisfy the system's constraints. In the first technique, called *static scheduling*, operations are scheduled statically on basic blocks, but constraints can span over several basic blocks and control-flow constructs. In the second technique, called *dynamic scheduling*, we dynamically schedule operations or parts of the design with respect to synchronization constraints, thus synchronizing the different parts of the specification with respect to each other.

The solution of both scheduling problems are cast as Integer Linear Programming instances and solved using Binary Decision Diagrams. There are two major advantages in using BDDs to solve ILP problems. BDDs are able to represent the whole space of solutions, instead of committing to a single solution that is obtained when using conventional ILP solvers. These solution space can be used later to better constraint other parts of the design. Moreover, we showed that BDDs allow us to consider a more compact constraint representation by encoding the design choices using Gray or binary encoding.

Results. We showed how the techniques presented in this thesis can be used to solve hard problems, such as the protocol conversion problem, the scheduling problem for program state machines, and the scheduling problem for specifications with exception handling mechanisms. We also contrasted our modeling and synthesis choices by showing the differences in the schedules obtained for BFSMs.

7.2 Future Work

During the development of this work, we observed that several choices were made that impacted the expressibility and synthesis results.

- Although using a model that includes concurrency, exception handling and variables increased the specification and synthesis capabilities, the assumption that every basic block executed in at least one cycle made by our approach compromised the quality of the synthesis for sequential models. It is worth researching a more general model where the one cycle constraint is imposed on loops only.

- We synchronized the concurrent parts of the design by dynamically scheduling operations over time. As the reader may have noted, the complexity of the dynamic scheduling technique may be too expensive at times. We believe that it is worth researching new techniques for synchronization synthesis by investigating automatic abstraction techniques for the specifications, or by developing new techniques to achieve synchronization synthesis.

- Since we used a model for analyzing the control-flow of the system by considering all possible schedules, and we presented a technique for incorporating design constraints from concurrent models, a natural extension to this is the verification of a system-level implementation. This could be achieved by considering our model to be the specification and the system's implementation to constraint the design. Thus, checking for a feasible implementation in this problem is equivalent to checking if the implementation satisfies the specification and the design constraints.

- The model we presented in our thesis can be extended to handle asynchronous events, which is necessary in order to consider systems consisting of hardware and software portions, that execute at different rates. In particular, software systems generally use a concurrency model better captured by interleaved concurrency. Thus, a model for hardware/software codesign should consider interleaved concurrency intermixed with the concurrency model presented for CFEs.

Bibliography

[Ake78] S.-B. Akers. Binary decision diagrams. *IEEE Transactions on Computers*, c-27, pp. 509–516, 1978.

[Bae90] J. C. M. Baeten. *Process Algebra.* Cambridge University Press, 1990.

[Bor88] G. Borriello. A new interface specification methodology and its application to transducer synthesis. UCB/CSD Technical Report (Dissertation) 88/430, U. C. Berkeley, 1988.

[BRB90] K. S. Brace, R. L. Rudell, and R. E. Bryant. Efficient implementation of a bdd package. In *Proceedings of the Design Automation Conference*, pp. 40–45, Orlando, FL, June 1990.

[Bry86] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, pp. 677–691, August 1986.

[Bry92] R. E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, pp. 293–318, September 1992.

[Brz64] J. A. Brzozowski. Derivatives of regular expressions. *Journal of the Association for Computing Machinery*, 11(4), pp. 481–494, October 1964.

[BS91] F. Boussinot and R. De Simone. The ESTEREL language. *Proceedings of the IEEE*, 79(9), pp. 1293–1303, September 1991.

[Cam76] R. H. Campbell. *Path Expressions: A Technique for Specifying Process Synchronization.* PhD thesis, University of Illinois, Urbana-Champaign. Department of Computer Science, August 1976. UIUCDCS-R-77-863.

[Cam91] R. Camposano. Path-based scheduling for synthesis. *IEEE Transactions on CAD/ICAS*, 10(1), pp. 85–923, January 1991.

[CBH+91] R. Camposano, R. A. Bergamaschi, C. E. Haynes, M. Payer, and S. M. Wu. The ibm high-level synthesis system. In R. Camposano and Wayne Wolf, editors, *High-Level VLSI Synthesis*, pp. 79–104. Kluwer Academic Publishers, June 1991.

[CE81] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. Technical Report TR-12-81, Harvard University, 1981.

[CHJ+90] H. Cho, G. Hachtel, S. Jeong, E. Schwarz, and F. Somenzi. Atpg aspects of fsm verification. In *Proceedings of the International Conference on Computer-Aided Design*, pp. 134–137, Santa Clara, November 1990.

[Cho74] Y. Choueka. Theories of automata on ω-tapes: A simplified approach. *Journal of Computer and System Sciences*, 8, pp. 117–141, 1974.

[CLR90] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms.* MIT Press, 1990.

[CM90] O. Coudert and J. C. Madre. A unified framework for the formal verification of sequential circuits. In *Proceedings of the International Conference on Computer-Aided Design*, pp. 126–129, Santa Clara, November 1990.

[DDT83] M. Davio, J.-P. Deschamps, and A. Thayse. *Digital Systems with Algorithm Implementation.* John Wiley & Sons, 1983.

[DGL92] A. Wu D. Gajski, N. Dutt and S. Lin. *High-Level VLSI Synthesis - Introduction to Chip and System Design.* Kluwer Academic Publishers, 1992.

[DH86] D. Drusinsky and D. Harel. Statecharts as an abstract model for digital control-units. Technical Report CS86-12, Weizmann Institute of Science, 1986.

[DKMT90] G. DeMicheli, D. C. Ku, F. Mailhot, and T. Truong. The olympus synthesis system for digital design. *IEEE Design and Test Magazine*, pp. 37–53, October 1990.

[FKJM93] D. Filo, D. C. Ku, C. N. Coelho Jr., and G. De Micheli. Interface optimization for concurrent systems under timing constraints. *IEEE Transactions on VLSI Systems*, 1(3), pp. 268–281, September 1993.

[FYDM93] J. Fron, J. Yang, M. Damiani, and G. De Micheli. Synthesis framework based on trace and automata theory. In *International Workshop on Logic Synthesis*, Tahoe, CA, May 1993.

[Geb91] C. H. Gebotys. *Optimal VLSI Architectural Synthesis.* Kluwer Academic Publishers, 1991.

[GG92] V. K. Garg and M. T. Gagunath. Concurrent regular expressions and their relationship to petri-nets. *Theoretical Computer Science*, 96(2), pp. 285–304, 1992.

[GJM92] R. K. Gupta, C. N. Coelho Jr., and G. De Micheli. Synthesis and simulation of digital systems containing interacting hardware and software components. In *Proceedings of the 29th Design Automation Conference*, pp. 225–230, June 1992.

[GJM94] R. K. Gupta, C. N. Coelho Jr., and G. De Micheli. Program implementation schemes for hardware-software systems. *IEEE Computer*, pp. 48–55, January 1994.

[GPR93] L. Guerra, M. Potkonjak, and J. Rabaey. High level synthesis for reconfigurable datapath structures. In *Proceedings of the International Conference on Computer-Aided Design*, pp. 26–29, November 1993.

[Gup93] R. K. Gupta. *Co-synthesis of Hardware and Software for Digital Embedded Systems.* PhD thesis, Stanford University, 1993.

[GVNG94] D. Gajski, F. Vahid, S. Narayan, and J. Gong. *Specification and Design of Embedded Systems.* Prentice Hall, 1994.

[HB84] K. Hwang and F. A. Briggs. *Computer Architecture and Parallel Processing.* McGraw-Hill Book Company, 1984.

[HLH91] C.-T. Hwang, J.-H. Lee, and Y-C Hsu. A formal approach to the scheduling problem in high-level synthesis. *IEEE Transactions on CAD/ICAS*, 10(4), pp. 464–475, April 1991.

[HLS] Benchmarks of the 1992 high-level synthesis workshop.

[HP82] L. Hafer and A. Parker. Automated synthesis of digital hardware. *IEEE Transactions on CAD/ICAS*, c-31(2), February 1982.

[HP92] Y.-H. Hung and A. C. Parker. High-level synthesis with pin constraints for multiple-chip designs. In *Proceedings of the Design Automation Conference*, pp. 231–234, June 1992.

[HU79] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison Wesley, 1979.

[Hu95] A. J. Hu. *Techniques for Efficient Formal Verification Using Binary Decision Diagrams*. PhD thesis, Stanford University, 1995.

[JS93] S.-W. Jeong and F. Somenzi. *Logic Synthesis and Optimization*, chapter A New Algorithm for 0-1 Programming Based on Binary Decision Diagrams, pp. 145–166. Kluwer Academic Plublishers, 1993.

[KD90] D. C. Ku and G. DeMicheli. HardwareC - a language for hardware design (version 2.0). CSL Technical Report CSL-TR-90-419, Stanford, April 1990.

[Keu89] K. Keutzer. Three competing design methodologies for asics: Architectural synthesis, logic synthesis, and module generation. In *Proceedings of the Design Automation Conference*, pp. 308–313, June 1989.

[Kle56] S. C. Kleene. Representation of Events by Nerve Nets In C. E. Shannon and J. McCarthy, editors, *Automata Studies*, pp. 3–42. Princeton University Press, 1956.

[KLL91] T. Kim, J. W. S. Liu, and C. L. Liu. A scheduling algorithm for conditional resource sharing. In *Proceedings of the International Conference on Computer-Aided Design*, pp. 84–87, Santa Clara, November 1991.

[KLMM95] D. Knapp, T. Ly, D. MacMillen, and R. Miller. Behavioral synthesis methodology for hdl-based specification and validation. In *Proceedings of the Design Automation Conference*, pp. 286–291, June 1995.

[KM92] D. Ku and G. De Micheli. *High-level Synthesis of ASICs under Timing and and Synchronization Constraints*. Kluwer Academic Publishers, 1992.

[KOH+94] J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. Gharachorloo, J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, and J. Hennessy. The Stanford FLASH multiprocessor. In *Proceedings of the International Symposium on Computer Architecture*, pp. 302–313, June 1994.

[Lan91] D. Lanneer et. al. Architectural synthesis for medium and high throughput signal processing with the new cathedral environment. In R. Camposano and Wayne Wolf, editors, *High-Level VLSI Synthesis*, pp. 27–54. Kluwer Academic Publishers, June 1991.

[Las90] J. Laski. Path expressions in data flow program testing. In 14^{th} *Anual International Computer Software and Applications Conference*, pp. 570–576, Chicago, 1990.

[LSU89] R. Lipsett, C. Schaefer, and C. Ussery. *VHDL: Hardware Description and Design.* Kluwer Academic Publishers, 1989.

[Mar90] P. Marwedel. Matching system and component behaviour in mimola synthesis tool. In *Proceedings of the European Design Automation Conference*, pp. 146–156, March 1990.

[McM93] K. McMillan. *Symbolic Model Checking.* Kluwer Academic Publishers, 1993.

[Mea55] G. H. Mealy. A method for synthesizing sequential circuits. *Bell System Technical Journal*, 34(5), pp. 1045–1079, 1955.

[Mic94] G. De Micheli. *Synthesis and Optimization of Digital Circuits.* McGraw Hill, 1994.

[Mil84] R. Milner. A complete inference system for a class of regular behaviours. *Journal of Computer and System Sciences*, 28, pp. 439–467, 1984.

[Mil85] G. J. Milne. Circal and the representation of communication, concurrency, and time. *ACM Transactions on Programming Languages and Systems*, 7(2), pp. 270–298, April 1985.

[Mil91] R. Milner. *Handbook of Theoretical Computer Science*, volume 2, chapter 19: Operational and Albebraic Semantics of Concurrent Processes, pp. 1201–1242. MIT Press, 1991.

[Mil94] G. Milne. *Formal Specification and Verification of Digital Systems.* McGraw-Hill, 1994.

[Moo56] E. F. Moore. Gedanken-Experiments on Sequential Machines, In C. E. Shannon and J. McCarthy, editors, *Automata Studies*, pp. 129–153. Princeton University Press, 1956.

[MPC90] M. McFarland, A. Parker, and R. Camposano. The high-level synthesis of digital systems. *Proceedings of the IEEE*, 78(2), pp. 301–318, February 1990.

[NEC93] *NEC Memory Products Data Book*, 1993.

[Nem88] G. Nemhauser. *Integer and Combinatorial Optimization.* John Wiley & Sons, 1988.

[NFCM92] S. Note, G. Goossens F. Catthoor, and H. De Man. Combined hardware selection and pipelining in high-performance data-path design. *IEEE Transactions on CAD/ICAS*, 11(4), pp. 413–423, April 1992.

[NG95] S. Narayan and D. Gajski. Interfacing incompatible protocols using interface process generation. In *Proceedings of the Design Automation Conference*, pp. 468–473, June 1995.

[NT86] J. Nestor and D. Thomas. Behavioral synthesis with interfaces. In *Proceedings of the Design Automation Conference*, pp. 112–115, June 1986.

[Pai77] M. R. Paige. On partitioning program graphs. *IEEE Transactions on Software Engineering*, SE-3(6), pp. 386–393, November 1977.

[PCI95] *PCI Local Bus Specification Revision 2.1*, 1995.

[Pet81] J. L. Peterson. *Petri Net Theory and the Modeling of Systems*. Prentice-Hall, 1981.

[RB93] I. Radivojević and F. Brewer. Symbolic techniques for optimal scheduling. In *Proceedings of the Synthesis and Simulation Meeting and International Interchange – SASIMI*, pp. 145–154, Nara, Japan, October 1993.

[RHSJ94] 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 Transactions on CAD/ICAS*, 13(2), pp. 167–177, February 1994.

[Rud74] S. Rudeanu. *Boolean Functions and Equations*. North-Holland, 1974.

[Rud93] R. Rudell. Dynamic variable ordering for ordered binary decision diagrams. In *International Workshop on Logic Synthesis*, Lake Tahoe, CA, April 1993.

[Sar89] R. Saracco. *Telecommunications Systems Engineering Using SDL*. Elsevier Science, 1989.

[Sea94] A. Seawright. *Grammar-Based Specification and Synthesis for Synchronous Digital Hardware Design.* PhD thesis, UC Santa Barbara, 1994.

[Sta70] E. Stabler. Microprogram transformations. *IEEE Transactions on Computers*, c-19, pp. 908–916, 1970.

[TM91] D. E. Thomas and P. R. Moorby. *The Verilog hardware description language.* Kluwer Academic Publishers, 1991.

[TSL+90] H. J. Touati, H. Savoj, B. Lin, R. K. Brayton, and A. Sangiovanni-Vincentelli. Implicit state enumeration of finite state machines using bdd's. In *Proceedings of the International Conference on Computer-Aided Design*, pp. 130–133, Santa Clara, November 1990.

[TU82] H. W. Trickey and J. D. Ullman. A regular expression compiler. In *IEEE COMPCON*, pp. 345–348, 1982.

[TWL95] A. Takach, W. Wolf, and M. Leeser. An automaton model for scheduling constraints. *IEEE Transactions on Computers*, 44(1), pp. 1–12, January 1995.

[VRB+93] J. Vanhoof, K. V. Rompaey, I. Bolsens, G. Goossens, and H. De Man. *High-level Synthesis for Real-time Digital Signal Processing.* Kluwer Academic Publishers, 1993.

[Wan88] T. H. Wang. *Repeatable Firing Sequences for Petri Nets under Conventional, Subset and Timed Firing Rules.* PhD thesis, Case Western Reserve University, 1988.

[Wol82] P. L. Wolper. *Synthesis of Communicating Processes from Temporal Logic Specifications.* PhD thesis, Stanford University, 1982.

[WT92] K. Wakabayashi and H. Tanaka. Global scheduling independent of control dependencies based on condition vectors. In *Proceedings of the Design Automation Conference*, pages pp. 112–115, June 1992.

[WTHM92] W. Wolf, A. Takach, C. Huang, and R. Manno. The Princeton university behavioral synthesis system. In *Proceedings of the* 29^{th}*Design Automation Conference*, pp. 182–187, June 1992.

[WTL91] W. Wolf, A. Takach, and T. Lee. *High-Level VLSI Synthesis*, R. Camposano and W. Wolf, editors, in *Architectural Optimization Methods for Control-Dominated Machines*. Kluwer Academic Publishers, 1991.

[YW] T.-Y. Yen and W. Wolf. Optimal scheduling for minimum dependence in fsms. *Accepted for publication in IEEE Transactions on VLSI Systems.*

[Zhu92] Z. Zhu. *Structured Hardware Design Transformations*. PhD thesis, Indiana University, 1992.

[ZJ93a] Z. Zhu and S. D. Johnson. Automatic synthesis of sequential synchronization. In *IFIP Conference on Hardware Description Languages and their Applications (CHDL '93)*, Ottawa, Canada, April 1993.

[ZJ93b] Z. Zhu and S. D. Johnson. An example of interactive hardware transformation. Technical Report 383, Indiana University, 1993.

[ZJ94] Z. Zhu and S. D. Johnson. Capturing synchronization specifications for sequential compositions. In *Proceedings of the International Conference on Computer Design*, pp. 117–121, October 1994. also as University of British Columbia TR 93-3.

Appendix A

Algebra of Synchronous Processes

In this section we define the algebra of synchronous processes (ASP), in a manner similar to the definitions found in [Bae90].

Definition A.1 *Let* $\mathbf{A} = (\Sigma_{ASP}, \delta, \epsilon, \gamma, E_{ASP})$ *be the model defined as follows.*

- *The alphabet of symbols of* Σ_{ASP} *denote the set of actions and the set of operations.*
  - *Actions are 0-ary functions and correspond to the atomic symbols.*
  - *Operations* $+, \cdot$ *and* $|$ *are binary functions and correspond to alternative, sequential and synchronous parallelism compositions, respectively.*
- *The two symbols* δ *and* ϵ *are defined as a deadlock and empty process, respectively.*
- *The function* γ *is a partial function defined as* $\gamma : \Sigma_{ASP} \cup \{\epsilon\} \times \Sigma_{ASP} \cup \{\epsilon\} \rightarrow \Sigma_{ASP} \cup \{\delta, \epsilon\}$. *One special symbol of* Σ_{ASP} *is denoted by 1 which is considered a unit element of* γ, *i.e.,* $\gamma(a, 1) = \gamma(1, a) = a$, *for all actions* a *of* Σ_{ASP}. *The empty process* ϵ *is considered a zero element of* γ, *i.e.,* $\gamma(a, \epsilon) = \gamma(\epsilon, a) = \epsilon$, *for all actions* a *of* Σ_{ASP}.

- *The set E_{ASP} defines the set of axioms for ASP.*

In order to define the axioms of ASP, we have to define the valid compositions as terms of ASP. Terms in the algebra of synchronous processes can be defined recursively as follows.

Definition A.2 *A term is the set defined as follows:*

- *δ and ϵ are terms.*
- *If a is a symbol from Σ_{ASP}, then a is a term.*
- *If x and y are terms, then $x \cdot y$ is a term.*
- *If x and y are terms, then $x + y$ is a term.*
- *If x and y are terms, then $x|y$ is a term.*

Nothing else is a term.

We denote generic terms of ASP by x, y and z, and generic actions by a and b. We can now present the axioms for ASP in Table 9.

$x + y = y + x$	A1
$(x + y) + z = x + (y + z)$	A2
$x + x = x$	A3
$(x + y) \cdot z = x \cdot z + y \cdot z$	A4
$(x \cdot y) \cdot z = x \cdot (y \cdot z)$	A5
$x + \delta = x$	A6
$\delta \cdot x = \delta$	A7
$x \cdot \epsilon = x$	A8
$\epsilon \cdot x = x$	A9
$a\|b = \gamma(a, b)$	CF1 and CF2
$a \cdot x\|b = (a\|b) \cdot x$	SC1
$a\|b \cdot y = (a\|b) \cdot y$	SC2
$a \cdot x\|b \cdot y = (a\|b) \cdot (x\|y)$	SC3
$(x + y)\|z = (x\|z) + (y\|z)$	SC4
$x\|(y + z) = (x\|y) + (x\|z)$	SC5
$x\|\epsilon = \epsilon$	SC6
$\epsilon\|x = \epsilon$	SC7

Table 9: *Axioms for ASP*

Appendix B

Binary Decision Diagrams

Binary Decision Diagrams [Bry86, Bry92] are a compact graph representation for Boolean functions based on the decision diagrams of Akers [Ake78]. A BDD represents a Boolean function by inferring all true and false values of the function by path traversals on the graph. A formal definition of BDDs due to Bryant [Bry86] is presented below.

Definition B.1 *A Binary Decision Diagram is a rooted, directed graph with vertex set V containing two types of vertices. A* **nonterminal** *vertex v has as attributes an argument index* $\texttt{index}(v) \in \{1, \ldots, n\}$ *and two children* $\texttt{low}(v), \texttt{high}(v) \in V$. *A* **terminal** *vertex v has as attribute a value* $\texttt{value}(v) \in \{0, 1\}$.

The correspondence between a BDD and a Boolean function can be seen from the following definition:

Definition B.2 *A BDD G having root vertex v denotes a function f_v, defined recursively as:*

1. *If v is a terminal vertex:*

   (a) *If* $\texttt{value}(v) = 1$, *then* $f_v = 1$.

(b) If $\texttt{value}(v) = 0$*, then* $f_v = 0$.

2. *If* v *is a nonterminal vertex with* $\texttt{index}(v) = i$*, then* f_v *is the function:*

$$f_v = x_i \, f_{\texttt{low}(v)} \vee \overline{x_i} \, f_{\texttt{high}(v)}$$

In addition to that, a BDD is **reduced** if $\texttt{low}(v) \neq \texttt{high}(v)$ for every vertex $v \in V$, and it is **ordered** if $\texttt{index}(v) < \texttt{index}(\texttt{low}(v))$ and $\texttt{index}(v) < \texttt{index}(\texttt{high}(v))$ for every nonterminal v.

Reduced ordered BDDs (ROBDDs) play an important role in Boolean function manipulation because they are canonical, i.e. if two BDDs are isomorphic, then they represent the same Boolean function. In this thesis, we will be referring to ROBDDs by BDDs.

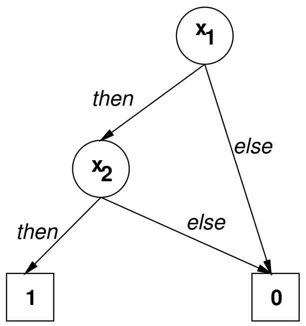


Figure 49: *Binary Decision Diagram for function* $x_1 x_2$

Example B.1.1. The function $f(x_1, x_2) = x_1 x_2$ can be represented by the BDD of Figure 49.

In this BDD, any path leading to vertex 1 represent an assignment to the variables that yields a *true* value for the Boolean function f, and a path leading to vertex 0 represent an assignment to the variables that yields a *false* value for the f. □

BDDs have been used in several different applications, including the solution of Integer Linear Programming [JS93] described earlier, because of its low space complexity to represent some types of Boolean functions. In these problems, each equation of the ILP problem is converted into a Boolean function [Rud74], which is represented by a BDD. This Boolean function contains all valid assignments to the Boolean variables that satisfy the ILP equation. Since in an ILP instance of the problem is specified as a set of equations and a cost function that should be minimized, an assignment satisfying this set of constraints can be obtained by conjoining the BDDs for the different equations, and a solution that minimizes a cost function can be obtained by a branch-and-bound on the set of valid assignments to the resulting BDD with respect to the cost function [JS93].

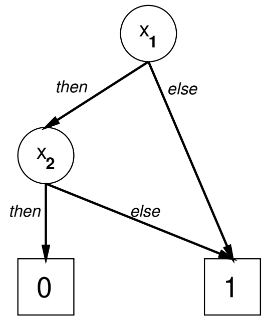


Figure 50: *BDD representing the constraint* $4x_1 + 5x_2 \leq 8$

Example B.1.2. 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 Figure 50. $\square$