An Algebra for Modeling Concurrent Digital Systems

Claudionor N. Coelho David Ku Giovanni De Micheli

Center for Integrated Systems
Stanford University
Stanford, California 94305
USA

Abstract

Most high-level synthesis approaches to date have focused solely on techniques for synthesizing circuits modelled as single processes. Realistic system designs are best modeled by multiple concurrent and interacting processes. In previous approaches, the interdependencies across process boundaries have been either ignored or addressed by ad hoc means. We propose in this paper an algebra called *control-flow expressions* that allows a modeling abstraction necessary for synthesis and analysis of synchronous, concurrent systems. In the algebra of *control-flow expressions*, functionality is abstracted by the execution time of the operations and by the control-flow of the specification. In addition to representing parallel computations and their synchronization, our formalism also serves as an unified framework to capture environment constraints, such as scheduling, binding, and synchronization requirements.

We use *decision variables* to represent the different choices in the control-flow structure of a specification. Those decision variables are derived from the original specification and the design constraints. Whereas conventional synthesis algorithms use a 0-1 assignment to the decision variables, no such restriction exists in our formalism, because we allow arbitrary Boolean functions to be assigned to decision variables.

We show an application of our formalism to the synthesis of synchronization among concurrent computations by means of an example.

1 Introduction

Whereas Boolean algebra and automata theory are the well-accepted theories for combinational and sequential logic synthesis, the existing formulations to the synthesis problems at higher-levels of specification are not general enough to capture the wider solution space of feasible designs. This opens a way to a host of specialized techniques for addressing specific aspects of concurrent designs, each with its own set of assumptions and limitations.

Two of the most common assumptions made by existing synthesis approaches are the *single-process* and the *static control-flow* assumptions. In the first case, each basic block of the control/data-flow graph is considered separately; the impact of synthesis decisions on the rest of the specification is either ignored, or addressed by specialized assumptions. In the latter case, the control-flow structure of the original specification is maintained throughout the synthesis flow. The emphasis is instead focused on exploiting the degrees of freedom in the data-flow via scheduling and binding approaches. The impact of the control-flow structure is again ignored or addressed by ad hoc techniques. Both assumptions result in localized synthesis decisions that can impede effective search of the global solution space.

For some application domains, such as digital signal processing, such localized approaches can yield efficient implementations. For control-dominated applications, however, these assumptions can result in overly conservative, even erroneous synthesis decisions because the interactions and interdependencies among concurrent computations over time are ignored. For example, existing approaches cannot systematically explore the sharing of critical resources across concurrent process and control-flow boundaries. Of fundamental importance is the rigorous and integrated analysis of concurrent computations, their control-flow structure, and their communication and synchronization.

We propose in this paper the algebra of *control-flow expressions* (CFE) as the basis for rigorous treatment of the relationships among control-flow structures in concurrent digital systems. The algebra has the following major characteristics:

- *Expressiveness.* The CFE model incorporates concurrency, synchronization, and control-flow. In addition, it also provides the basis for uniformly capturing *constraints* on the specification. Furthermore, CFEs allows the abstraction of the *synchronization* among concurrent computations, a crucial prerequisite to the formal treatment of don't cares in multi-process synthesis.
- *Rigorous foundation.* The CFE formalism is based on an extension of the algebra of regular expressions to incorporate guards on computations. The manipulation and analysis of control-flow expressions are therefore based on solid theoretical basis. For the purposes of this paper, however, we will give a more informal definition of the algebra, deferring the formal treatment to a later paper.
- *Systematic solution strategy.* We show the use of *decision variables* in the CFE model to capture both the control-flow structure as well as the degrees of freedom due to the imposed constraints. A solution strategy can be obtained by assigning Boolean expressions to the decision variables.

The abstraction level provided by control-flow expressions allows for the representation of both the control-flow and the synchronization constructs of hardware description languages, such as Verilog, VHDL and HardwareC. The key is again on a rigorous formalism to represent, analyze, and manipulate concurrent, control-dominated systems.

The following example shows the importance of analyzing the interdependencies among concurrent computations.

Example 1 Figure 1 shows three processes p_1, p_2 and p_3 that communicate with a memory through a single shared bus. For example, these three processes can be a model of the specification of a DMA controller. In this figure, O_1 and O_2 denote abstractions on some single-cycle operation in the original specification. Each process restarts execution upon completion of its operations. The bus requirements of the processes are such that process p_1 accesses the memory bus every two clock cycles; bus access by process p_2 requires synchronization with signal c; and the synchronization on process p_3 is denoted by the operation *wait*.

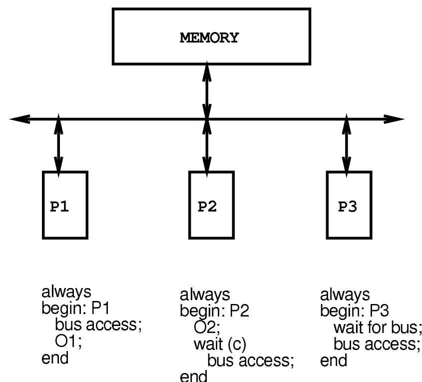


Figure 1: *Example of a Concurrent Digital System*

Assume p_1 and p_2 have already been synthesized. We want to analyze the interdependencies over time among p_1, p_2 and p_3 so that this information can be used to synthesize p_3 and its synchronization. We will show how CFEs can be used to model the processes, and how the problem of synthesizing the synchronization to avoid bus conflicts can be solved in our algebraic formalism. □

The organization of the paper is as follows. First, we describe representative research efforts in this field. We present in Section 3 the algebra of control-flow expression, and how computations are abstracted by actions and guards. In Section 4, we show how scheduling, binding, and synchronization constraints are represented by control-flow expressions. Finally, we show by means of an example the synthesis of synchronization for concurrent processes. Specifically, we demonstrate how this problem can be solved if one allows the assignment of arbitrary Boolean functions to the decision variables.

2 Related Research

Two models for the execution semantics of computations have been defined for the analysis and verification of concurrent systems - the interleaved execution semantics and the synchronous

execution semantics. In the interleaved execution model, there is no precise timing information about the execution of the concurrent computations (also called processes in this paper). This execution model is non-deterministic in the execution of operations either by assuming that each operation of the process take an unbounded amount of time to complete or by having an unknown delay between sequential operations. Process Algebras [1], CSP [8], CCS [13], trace theory [15, 4] and Petri nets [16] are theories based on this execution semantic. Although this model is very general and have been used in the modelling of a large class of problems, the complexity for the analysis of concurrent computations under the interleaved model can be very high.

In the synchronous execution semantics, also called the maximal parallelism model, time is well defined. This means that the execution time of operations can be specified in terms of a global clock, and that operations are executed as soon as possible. This model is more restrictive than the interleaved model, but since digital synchronous concurrent systems are implemented as a set of concurrent finite-state machines under a single clock, this model is more appropriate. The *Algebra of Synchronous Processes* [1], and the systems defined in [12, 5] are examples of this model.

The use of the synchronous execution semantics for control-flow expressions was based on the following two reasons. First, its complexity can be better controlled than the complexity of the interleaved model, since the later can be simulated in the synchronous model if necessary. Second, further simplification can be achieved by considering the implicit synchronization (with respect to the single clock) among the concurrent parts of the description.

Control-flow expressions, can be considered as a subset of the *Algebra Synchronous Processes*, restricted to regular processes. When compared to Petri-Nets, a correspondence can also be made (see [16]) if synchronous firing rules are used.

In the area of high-level synthesis, several approaches are based on integer-linear programming (ILP) formulations [6, 9, 7]. Although exact solutions can be found by solving the ILP formulation, these approaches are usually restricted to either the single-process or static control-flow assumptions. Trickey [14] was the first to address the problem of control-flow restructuring during synthesis, but this system relied on the single process assumption. Wolf proposed *behavioral FSMs* (BFSM) as an automata theoretic paradigm for synthesis, where a BFSM is transformed into register-transfer FSMs based on the imposed sequencing and timing constraints [17]. Control flow expressions can be considered as a higher level of abstraction than the level of abstraction of BFSMs - constraints and bindings can be expressed in terms of control-flow expressions, but no such representation exists in terms of BFSMs. This uniform representation of CFEs pays off when solving problems that involve a conditional selection of a path in terms of the design constraints. Interface matching [10] takes advantage of the strong coupling between concurrent processes to reduce the synchronization and communication costs. Although this approach can handle complex timing constraints across process synchronizations, it is restricted by the static communication and control-flow assumption. In the case of CFEs, the interactions among concurrent computations is more general, because it is not restricted to basic blocks.

In summary, the algebra of control-flow expressions integrates the modeling of control-flow with the abstraction of synchronizations among concurrent computations. It provides an unified framework for capturing both design behavior and constraints, along with rigorous solution strategy, to address the analysis and synthesis of concurrent digital systems.

3 Control-Flow Expressions

We informally present in this section the algebra of *control-flow expressions* (CFEs) as an extension of regular expressions with the addition of guards on actions, parallel composition, and an execution time semantic for computations. Although CFEs can be also defined in terms of the *Algebra of Syncrhonous Processes*, we define CFEs as an extension of regular expressions because it is more succinct.

In the CFE model, operations in a high-level specification are abstracted in the form of *actions*, and the conditions on alternative and looping constructs are abstracted by means of *guards*. CFEs

abstract the computation of a set of operations in a high-level specification. Actions represent the finest grain in our computational model. For the sake of simplicity, we assume that every action executes within one unit delay (e.g., one cycle). Multi-cycle operations can be represented by the sequential composition of unit cycle actions.

The algebra of control-flow expressions can be formally defined by the triple (Φ, Γ, ϵ), where Φ denotes the set of variables, Γ denotes the set of operators or functions on control-flow expressions, and ϵ denotes a null element on the control-flow expression domain, which executes within zero time.

The set of variables Φ can be further subdivided, as follows:

- A set of action variables $\mathcal{A}$ is associated with each operation,
- A set of function variables $\mathcal{F}$ is associated with each CFE, representing an abstraction of the CFE computation,
- A set of variables $\mathcal{C}$, called conditionals, is associated with guards.

Each variable $v \in \Phi$ represents the abstraction of some functionality in the original specification. A variable representing a control-flow expression allows the definition of a computation hierarchically. At the lowest level in the hierarchy, we define an action to be a CFE that executes atomically, i.e. within one clock cycle and representing a single computation in the original specification. One action that is always included in the set of variables is the no-operation action, denoted by 0, which executes in one clock cycle. It is used either when some delay must be inserted between two actions in a specification or when the operation performed by an action is not relevant to the other concurrent parts of the specification and therefore its details can be hidden. The difference between 0 and ϵ is that 0 actually represents computation that remains idle for one cycle. Executing an ϵ immediately transfers control without taking any time.

Guard variables represent the conditions under which sets of operations are executed in a control-flow expression. They are associated with the conditions that enables alternative and looping paths to be activated. When the guards represent conditions of the original specification, they will be called *conditionals*, which are represented by $\mathcal{C}$. In the next section, we will introduce another type of *guard* variables, named *decision variables*.

More generally, since guard variables are Boolean variables, we allow the composition of those variables in Boolean expressions. Those expressions will be called in this paper *guarded expressions*, and they will be used as guards for CFEs.

Example 2 As mentioned before, computations in the original specification are abstracted in terms of control-flow expressions and guards. In the specification *if* $(x == 3)$ $z = u + w$, the operation $z = u + w$ can be abstracted by the action variable a and the condition $x == 3$ can be abstracted by the guard c. The resulting CFE representing the description above is given by $c : a$. $\square$

We now define the set Γ of operations and functions for composing control-flow expressions in our algebraic model, as follows:

Definition 3.1 *Let* $p, q \in \mathcal{A}$ *and* c_1, c_2 *be guarded expressions. Then, control-flow expressions can be composed by the rules* $(p.q)$, $(p||q)$, $(c_1 : p + c_2 : q)$, $(c_1 : p)^*$ *and* p^ω.

We assume in this paper that the precedence of the operators is as follows: $*, \omega > : > . > + > ||$. This precedence may be overruled by parenthesis. Also, whenever two actions a and b execute at the same cycle for exactly one clock cycle, we represent those actions as $\{a, b\}$.

All compositional rules of CFEs presented above have their counterpart in high-level descriptions and sequencing graphs [11]. This facilitates the interaction between CFEs and higher levels of abstractions. The operator $p.q$ defines the sequential composition, meaning that p is executed before q is executed. Operator $c_1 : p + c_2 : q$ defines a general form of an alternative construct, meaning that either p or q is executed, but not both, depending whether c_1 or c_2 is true. Operator $p||q$ is defines the parallel execution of p and q. Operator $(c_1 : p)^*$ defines a looping construct, meaning that whenever c_1 is true, p is executed. Operator p^ω defines the infinite computation or iteration of p. Upon reset, p will be repeated infinitely many times. The table in Figure 2 represents the equivalences between Verilog constructs and CFEs.

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

Figure 2: *Link between Verilog Constructs and Control-Flow Expressions*

```
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 3: *Greatest-Common Divisor Example*

Example 3 We provide here an example on the representation of Verilog constructs in control-flow expressions. The specification shown in Figure 3 consists of an algorithmic representation of a greatest common divisor, in some high-level hardware description language. Its control-flow expression is presented below, where the labels on the left correspond to the actions being executed or the conditionals on alternative compositions.

$$GCD = [(r : 0)^*.b.(c1 : (c2 : (c3 : c)^*.d)^*.e + \tilde{}c1 : \epsilon)]^\omega$$

□

3.1 Operational Semantics of CFEs

We present here a series of relations among CFEs that will allow the modification of the original control-flow expression in accordance with the original specification and constraints. We also present the rule *compose_in_parallel* that allows the investigation of synchronization, communication and implicit coordination among the parallel parts of the specification.

The following equivalence relations can be considered an extension of ω-regular expressions [2, 3], and will be used throughout in this paper.

$$\begin{aligned} p^\omega &\Leftrightarrow p.p^\omega \\ (c : p)^* &\Leftrightarrow (c : p).(c : p)^* + \tilde{}c : \epsilon \\ c : p + \tilde{}c : p &\Leftrightarrow p \\ c_1 : p_1.s + c_2 : p_2.s &\Leftrightarrow (c_1 : p_1 + c_2 : p_2) : s \end{aligned}$$

The first two relations unroll the infinite and looping computations, respectively. The following two relations allow the reduction of the CFE by merging similar sub-computations. These rules are used in the examples that follow.

In order to analyze the synchronization among processes, we need to compose the specifications in parallel. This can be achieved by the rule *compose_in_parallel*, which eliminates the || operator in a control-flow expression. In order to define the semantics of the rule *compose_in_parallel*, we need the following definitions:

Definition 3.2 *If the CFE $p = p_1.p_2$, where p_1 and p_2 are also CFEs, we call p_1 a* **prefix** *of p, and p_2 a* **suffix** *of p.*

Expression p_1 is defined as a **proper prefix** *of p if $p_2 \neq \epsilon$. Similarly, expression p_2 is defined as a* **proper suffix** *of p if $p_1 \neq \epsilon$.*

Definition 3.3 *Let $\sum$ denote the* **alternative** *composition of CFEs. If the CFE $p = (\sum_i c_i :$*

$p_i).q$, *where* p_i *and* q *are CFEs, we say that* c_i *is the guard of the* **alternative composition** $p_i.q$

Definition 3.4 *Let* p *and* p' *be control-flow expressions, such that* p' *is a proper suffix of some alternative composition of* p*, guarded by* c*. If* a *is the first action of* p *executed in the path guarded by* c*, then* $p \stackrel{c:a}{\rightarrow} p'$ *represents that after* a *is first executed in* p *when guard* c *is true, the actions in* p' *must be executed next.*

Definition 3.5 *A* **path** *of a CFE* p *is a sequential composition of actions* $a_1.a_2.\cdots.a_n$ *such that* $\exists$ *suffixes* $p_1, p_2, \ldots, p_n$ *of* p *and* $p \stackrel{c_1:a_1}{\rightarrow} p_1 \stackrel{c_2:a_2}{\rightarrow} \ldots \stackrel{c_n:a_n}{\rightarrow} p_n$.

We can now define the *compose_in_parallel* rule:

Definition 3.6 *Let* p,q,p' *and* q' *be CFEs,* c_1 *and* c_2 *guards, and* a *and* b *action sets (or only actions in the case of singleton sets). Then, compose_in_parallel eliminates the* $||$ *operation by applying the following rule iteratively:*

$$(p \stackrel{c_1:a}{\rightarrow} p') \wedge (q \stackrel{c_2:b}{\rightarrow} q') \Leftrightarrow (p||q) \stackrel{(c_1 \wedge c_2):\{a,b\}}{\rightarrow} (p'||q')$$

Intuitively, if the guard c_1 of the alternative composition $a.p'$ of p is *true* and the guard c_2 of the alternative composition $b.q'$ of q is also *true*; and if p and q are executing in parallel (represented by $(p||q)$), then c_1 and c_2 are *true*, actions a and b are executed in parallel, and the CFEs p' and q' are also executed in parallel. Note that we assumed that a and b executed within one cycle.

The following example presents an application of the rule *compose_in_parallel* shown above.

Example 4 In this example, we show how the *compose_in_parallel* can be used in the analysis of concurrent systems in hardware.

In the CFE $p = (a.b.c)^\omega||(d.e)^\omega$, assume b and d are adders and we want to use the same component to implement both b and d. In order to do that, actions b and d must be non-conflicting actions. This can be checked by composing both parallel parts of the specification together, and then traversing the resulting expression in order to look for any action set which includes $\{b, d\}$. If such an action set exists, then b and d can not share the same adder.

If we apply the algorithm *compose_in_parallel* to p, we obtain:

$$\begin{aligned} q &= \textit{compose_in_parallel}(p) \\ &= (\{a,d\}.\{b,e\}.\{c,d\}.\{a.e\}.\{b,d\}.\{c,e\})^\omega \end{aligned}$$

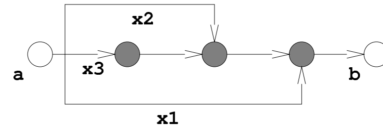
We can see that in the fifth clock cycle b and d execute at the same time, so any implementation with this schedule would require a different adder for b and d in order to avoid the conflict. □

4 Constraints in Control-Flow Expressions

In general, specifications contain constraints such as time slacks between two operations, binding constraints or even synchronization constraints. Those constraints are used during the synthesis and analysis of a specification. By extending CFEs to capture the specification and the constraints, we create a framework that can be used for both synthesis and analysis of a specification. In this section, we show how these constraints can be uniformly incorporated.

We consider here three types of design constraints: synchronization constriants, timing or scheduling constraints, and bindingconstraints.

Synchronization constraints are incorporated by defining allowed and forbidden sets of actions, here called *ALWAYS* and *NEVER* sets. Those sets of valid actions can be checked during the application of the rule *compose_in_parallel* to the specification, and the paths that do not observed those constraints of allowed and forbidden sets are eliminated.



a.(x1:0 + x2:0.0 + x3:0.0.0).b

Figure 4: *CFE for timing constraint of* $[2, 4]$ *between* a *and* b

Timing constraints are incorporated in the specification by composing the specification in parallel with a CFE that specifies the constraint. A *maximum* timing constraints between two actions a and b is specified by the CFE $a.(\sum x_j : 0^j).b$,

where x_j is a decision variable, 0^j is 0 repeated j times (or a slack of $j+1$ cycles between a and b), and j is less than the maximum number of cycles allowed. In this expression, when x_j is different from 0 (or in other words, when the path is enabled x_j), there will be a delay of $j+1$ cycles between a and b. A *minimum* timing constraint between two actions a and b is represented by the CFE $a.0^n.(x : 0)^*.b$, meaning that the delay between a and b should be at least $n+1$ cycles. Figure 4 presents an example of a minimum timing constraint of 2 cycles and a maximum timing constraint of 4 cycles between two actions a and b. The hachured nodes denote delay elements inserted between the nodes that represent actions a and b. In this figure, x_1, x_2 and x_3 denote the decision variables guarding the alternative paths of the design.

Binding constraints are incorporated into the system by rewriting actions as guarded expressions in the original CFE. For example, an action a that could be implemented by components i or j would be rewritten as $(x_{ai} : i + x_{aj} : j)$, where decision variable x_{ai} being equal to 1 indicates that action a is implemented by component i and mutatis mutandis for component j.

5 Synchronization of Concurrent Digital Systems

In this section we provide a small full example on the application of CFEs in multi-process synthesis problems and show how the assignment of Boolean expressions to decision variables allows the solution space to be greatly enhanced. Consider the problem defined in Example 1, i.e. we want to synthesize the synchronization for a process with respect to some resource utilization of other processes.

The three concurrent specifications p_1, p_2 and p_3 can be written as CFEs, as follows:

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

Where a is the critical resource (the bus or memory port). Note that by saying that all transactions are made through a single bus, we are in fact pre-binding the transfers to the same bus.

Assume that p_1 and p_2 have already been synthesized and that the conflicts of p_1 and p_2 are resolved elsewhere (by the input patterns of the conditional c, for example). Our problem is the synthesis of the synchronization between p_3 and the combination of p_1 and p_2 such that p_3 does not attempt to use resource a when either p_1 or p_2 is using it.

Let us first rewrite p_3 as $[(x : 0)^*.a]^\omega$. Here, $wait_a$ was substituted into the synchronization $(x : 0)^*$, where x is a decision variable. The problem of finding a synchronization for p_3 is then equivalent to finding an assignment of x over time such that p_3's use of a does conflict with p_1 and p_2's use of a.

In order to do that, we first eliminate the $||$ operation from $p = p_1 || p_2 || p_3$, with the constraint that $ALWAYS = \emptyset$ and $NEVER = \{a, a\}$, which gives the following expression:

$$\begin{aligned} p = \; & [(x : a).(((c \wedge x) : 0 + (c \wedge \tilde{\ }x) : a). \\ & ((c \wedge x) : a))^*.((\tilde{\ }c \wedge x) : a)]^\omega \end{aligned}$$

This expression contains all possible implementations that satisfies the constraint that no two transfers use the bus at the same time. A particular implementation can be obtained by assigning the decision variables to Boolean functions over the *guard variables*. This assignment to the decision variables will be called a solution to the synthesis problem. Note that p can be solved by a procedure that searches through the valid paths of p. For example, in p, the only choice that needs to be considered is when c is true in the loop, since in this case x could take either a value of 1 or 0. In this case, x needs to be assigned to 0, since the other case represents the starvation of p_3. By just assigning x to 0 to the decision variable x is not enough to guarantee that p_3 does not have any conflicts with p_1 and p_2 - just look that in the next cycle, x should be assigned to 1. Thus, what we need is to assign x to 0 when the other guard variables enable this path, or in this case, assign x to 0 when c is true, which is equivalent to assigning x to $\tilde{\ }c$ at this control-step.

Now, by considering the assignments of the decision variables in p, and the actions defined in p_3, we can reconstruct p_3 as:

$$p_3 = [0.(\tilde{~}c : 0.0)^*.a]^\omega$$

This corresponds to the correct behavior, since p_3 is only able to use resource a starting at the second clock cycle, spaced by 2 (from p_1) and when c is false, meaning that the second process is still executing the loop.

6 Conclusions

We have proposed an algebraic formulation for the modeling and synthesis of concurrent systems in hardware using the maximal parallelism semantics, called control-flow expressions (CFEs). This algebra was based on concurrency and synchronization among the processes, which is of utmost importance in the synthesis of multi-process systems.

We also showed how design constraints in terms of synchronization, scheduling and binding could be incorporated into the design. Since the constraints could be cast in terms of CFEs, we showed that CFEs could be used as a unified framework for specification abstraction and incorporation of constraints.

One example on the utilization of CFEs in a multi-process synthesis environment was provided. The use of decision variables as guards of alternative paths and the analysis of the implicit synchronization among concurrent processes allowed the solution of the synchronization synthesis problem. This was done by assigning Boolean functions to the decision variables.

We have developed a solution algorithm for assigning decision variables in CFEs. Due to the lack of space, this algorithm will be presented in a future paper.

7 Acknowledgements

This research was sponsored by NSF/ARPA, under grant No. MIP 9115432, and by WESTERN UNION NSF No. INT-9123222. The first author was supported by CNPq-Brazil under contract 200212/90.7.

References

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

[2] J. A. Brzozowski. Derivatives of regular expressions. *Journal of the ACM*, 11(4), October 1964.

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

[4] D. L. Dill. Trace theory for automatic hierarchical verification of speed-independent circuits. Technical Report CMU-CS-88-119, Carnegie Mellon University, February 1988.

[5] V. Akella G. C. Gopalakrishnan, N. S. Mani. *Formal VLSI Specification and Synthesis, VLSI Design Methods*, volume I, chapter A Design Validation System for Synchronous Hardware Based on a Process Model: A Case Study, pages 227–246. Elsevier Science Publishers B.V. (North-Holland), 1990.

[6] C. H. Gebotys and M. I. Elmasry. A global optimization approach for architectural synthesis. In *Proceedings of the International Conference on Computer-Aided Design*, pages 258–261, Santa Clara, CA, November 1990.

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

[8] C. A. R. Hoare. A model for communicating sequential processes. Technical report, Oxford University, 1981.

[9] 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):464–475, April 1991.

[10] D. Ku, D. Filo, C. Coelho, and G. De Micheli. Interface optimization for concurrent systems under timing constraints using interface matching. In *High Level Synthesis Workshop*, November 1992.

[11] David Ku and Giovanni De Micheli. *High-level Synthesis of ASICs under Timing and and Synchronization Constraints*. Kluwer Academic Publishers, 1992.

[12] L. Y. Liu and R. K. Shyamasundar. Static analysis of real-time distributed systems. *IEEE Transactions on Software Engineering*, 16(4):373–388, April 1990.

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

[14] H. Trickey. Flamel: A high-level hardware compiler. *IEEE Transactions on CAD/ICAS*, CAD-6:259–269, March 1987.

[15] J. L. A. van de Snepscheut. *Trace Theory and VLSI Design*. Springer Verlag, 1985.

[16] R. J. van Glabeek and F. W. Vaandrager. *Lecture Notes in Computer Science 259*, chapter Petri net Models for Algebraic Theories of Concurrency, pages 224–242. Springer-Verlag, 1987.

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