A Language for Compositional Specification and Verification of Finite State Hardware Controllers

E. M. Clarke, D. E. Long, K. L. McMillan

School of Computer Science, Carnegie Mellon University
Pittsburgh, PA 15213, U. S. A.

Abstract

SML is a language for describing complex finite state hardware controllers. It provides many of the standard control structures found in modern programming languages. The state tables produced by the SML compiler can be used as input to a temporal logic model checker that can automatically determine whether a specification in the logic CTL is satisfied. We describe extensions to SML for the design of modular controllers. These extensions allow a compositional approach to model checking which can substantially reduce its complexity. To demonstrate our methods, we discuss the specification and verification of a simple CPU controller.

[0] This research was sponsored in part by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 4976, Amendment 20, under Contract Number F33615-87-C-1499, monitored by the:

Avionics Laboratory
Air Force Wright Aeronautical Laboratories
Aeronoautical Systems Division (AFSC)
United States Air Force
Wright-Patterson AFB, Ohio 45433-6543

The National Science Foundation also sponsored this research effort under Contract Number CCR-8722633. The second author is supported by an NSF graduate fellowship.

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the US Government.

The programming language CSML[1] provides a concise notation for specifying complicated hardware controllers [4, 5, 6]. It has many of the control structures found in modern imperative programming languages including a while statement, a conditional statement, a case statement, and a parallel execution statement. There is even a simple mechanism for declaring non-recursive procedures. However, it differs from standard programming languages in two important respects. First, the passage of time is explicit in the semantics of CSML; like Esterel, CSML is based on a synchronous model of time. The actions of a program are presumed to be instantaneous, and time is measured by events external to the program. Second, all CSML programs are finite state, hence programs in CSML may be compiled into state transition tables that can be implemented in hardware as PALs, PLAs, or ROMs. In addition, the state transition tables can be used as input to a *temporal logic verifier* that allows various safety and liveness properties of programs to be verified automatically.

The compilation of control programs into a state transition tables is also useful for real-time systems since most of the computation is done at compile time, and programs can be implemented in hardware using standard, regular structures. A major disadvantage of this technique is the possibility of an explosion in the number of states of the controller. This problem typically occurs in controllers with several loosely coupled parallel processes, where the number of reachable states is exponential in the number of processes. The problem can be at least partially avoided by writing programs in a compositional or hierarchical manner. Use of such design techniques results in an implementation composed of a number of relatively small state machines and may allow automatic verification with respect to a reduced model.

Figure 1 illustrates the process of implementing and verifying a controller. The CSML compiler generates state tables for a collection of modules (Moore machines) that can be implemented directly in hardware. The modules can also be composed to generate a global state table. Using a theorem called the *interface rule*, the state explosion involved in this composition can be reduced by first hiding some signals, and then minimizing the modules. This results in a reduced global state table which preserves the truth value of a subset of formulas in the temporal logic CTL. The specification of the program in CTL is then verified with respect to this reduced model, using an efficient model checking algorithm [2, 9]. If any formula in the specification is false, the model checking program will generate an execution trace (provided such a trace exists) that is a counterexample to the formula. The counterexample trace is generally quite useful for finding and fixing errors in the controller.

Effective use of these compositional techniques requires a design style with well defined interfaces that hide the underlying complexity of subsystems. To illustrate this design style, and the compositional model checking method, we describe the design and verification of the controller for a simple CPU with decoupled access and execution units. We show how the controller can be implemented in CSML and provide a formal specification in CTL of one of the two modules. We then demonstrate how to apply the interface rule to reduce the complexity of automatically verifying this specification. In this example, using the interface

[1]Compositional State Machine Language

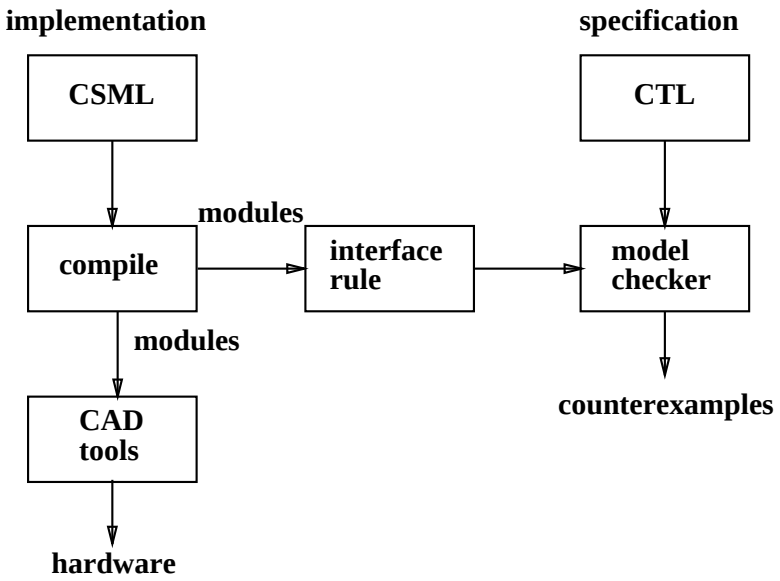

Figure 1: Flowchart of the compositional model checking technique.

rule reduces the number of states by approximately a factor of 6.

Section 1 of this paper introduces the logic CTL and the interface rule. The SML language on which CSML is based is described briefly in section 2, and the CSML dialect is covered in section 3. Finally, section 4 discusses the CPU example and the results of the model checking procedure.

1 The logic

The logic we use for formal specification is a branching-time temporal logic called CTL [8]. Formulas in CTL are built from atomic propositions (the signals of the system), boolean connectives ($\wedge$, $\vee$, $\rightarrow$ and $\neg$), and temporal operators which are used to specify timing relationships.

We view one of the Moore machines produced by the CSML compiler as an infinite computation tree. Each node in the tree corresponds to a global state of the system from the Moore machine's point of view, and hence consists of a state of the Moore machine and a valuation of the machine's inputs. At the next clock event, the Moore machine will make a transition to a new state and the inputs may change to an arbitrary value. The new state of the Moore machine plus a new input valuation represents a possible successor state in the computation tree. An infinite sequence of nodes, each a successor of the previous one, is called a path. Each path represents a possible computation of the system starting from some initial state. We will also refer to nodes in the tree as states.

Each temporal operator in our logic consists of a path quantifier ($\forall$ or $\exists$) and a modality (F, G, X, or $\mathcal{U}$). The modality specifies a temporal property of paths, while the quantifier

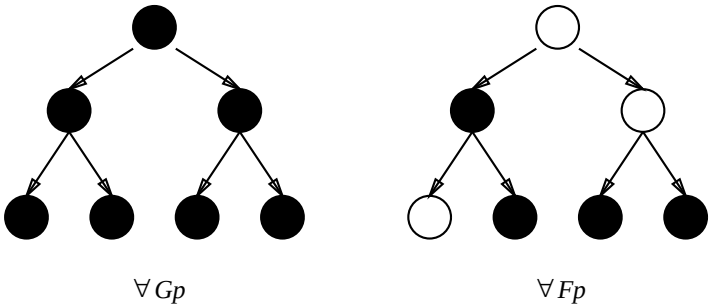

Figure 2: Basic temporal operators

specifies that the property must hold for all paths or for some paths beginning with a given state. The following are the modalities we can specify in CTL.

i. $F\varphi$ means that φ is true at some point in the future.

ii. $G\varphi$ means that φ holds in the present and at all points in the future.

iii. $X\varphi$ means that φ is true at the next state.

iv. $\varphi \mathcal{U} \psi$ means that ψ holds at some point in the future, and that until that point, φ is true.

v. $\varphi \mathcal{W} \psi$ means that either $G\varphi$ is true or $\varphi \mathcal{U} \psi$ is true.

Figure 2 shows two of the basic temporal operators, each with an example computation tree for which the operator is true. The solid nodes in the trees represent states where the atomic proposition p holds; in the open nodes, $\neg p$ holds.

The following two examples illustrate the expressive power of the logic.

i. $\forall G(req \rightarrow \forall F ack)$ specifies that along every path, if the signal *req* occurs, then eventually *ack* occurs also.

ii. $\forall G(send \rightarrow \forall(send \,\mathcal{U}\, rcvd))$ states that along every path, if *send* occurs, then *rcvd* must eventually occur and *send* must remain asserted until *rcvd* occurs.

Given a finite state machine and a formula in a temporal logic, the model checking problem is to find all the states where the formula is true. Clarke *et al* [9] give an efficient graph-traversal algorithm to solve the model checking problem for the logic CTL. For example, to determine if the formula $\forall Fp$ is true in state s, we look for an infinite path starting at s such that p is false at each state on the path. If no such path exists, the formula is true at s; otherwise it is false. In a finite Kripke structure, an infinite path from s along which p is false in every state exists only if there is a loop along which p is always false, and the loop is reachable from s via a path where p is always false. If such a loop exists, it can be found by computing the strongly connected components of the state graph after removing

all of the states in which p is true. If there exists a strongly connected component in this graph reachable from s, then the formula $\forall Fp$ is false, and we can exhibit a looping path which demonstrates that the formula is false. A counterexample path of this sort can be of help in locating the source of an error in a complex finite state system.

The CTL model checker allows the specification of fairness constraints. A fairness constraint restricts the computation tree to those paths along which a certain formula holds infinitely often. This type of restriction is commonly used to represent assumptions about fair scheduling in a concurrent system. For example, to prove that a process eventually terminates *assuming* it is allowed access to some resource infinitely often, we would use a fairness constraint.

We deal with the state explosion in concurrent systems using the *interface rule* to reduce the number of states. The idea is to form simple abstractions of the modules in the system and to use these abstractions when building a state graph for the model checker. Figure 3 illustrates the principle. In this figure, P_1 and P_2 represent the components of the system we wish to reason about. The components are connected by a set of signals S. Suppose we want to determine whether P_2 in the context of the entire system satisfies some property. In order to do this, we could compose P_2 in parallel with the environment P_1, but this may result in a model with a very large number of states. Instead, we consider how P_2 interacts with the environment. Intuitively, since P_2 can only observe the environment via the the signals in S, replacing P_1 with any process A_1 which is equivalent to P_1 with respect to the signals in S will result in equivalent observable behavior of P_2. If A_1 is smaller than P_1, we have reduced the complexity of the verification problem.

The interface rule formalizes the above line of reasoning. To use the interface rule, the state machine A_1 must be equivalent ($\equiv$) to P_1 on S in an appropriate sense, which preserves the truth value of logical formulas in the composition. For the logic CTL and systems composed of Moore machines, the ordinary notion of Moore machine equivalence is sufficient [10]. In the verification process, when we want to verify that P_2 in the system satisfies a property, we will take P_1 and hide all its outputs except for those in S. We then apply the standard Moore machine minimization algorithm to obtain A_1. After composing A_1 and P_2 and checking the desired property, we use the interface rule. This rule states that if:

i. $P_1 \equiv A_1$ on the set S,

ii. φ is a CTL formula whose atomic propositions denote signals of P_2, and

iii. φ is true in $A_1 \parallel P_2$ (the composition of A_1 and P_2),

then φ is true in $P_1 \parallel P_2$. In a loosely coupled system, A_1 will almost always have far fewer states than P_1, and thus $A_1 \parallel P_2$ will be much smaller than $P_1 \parallel P_2$.

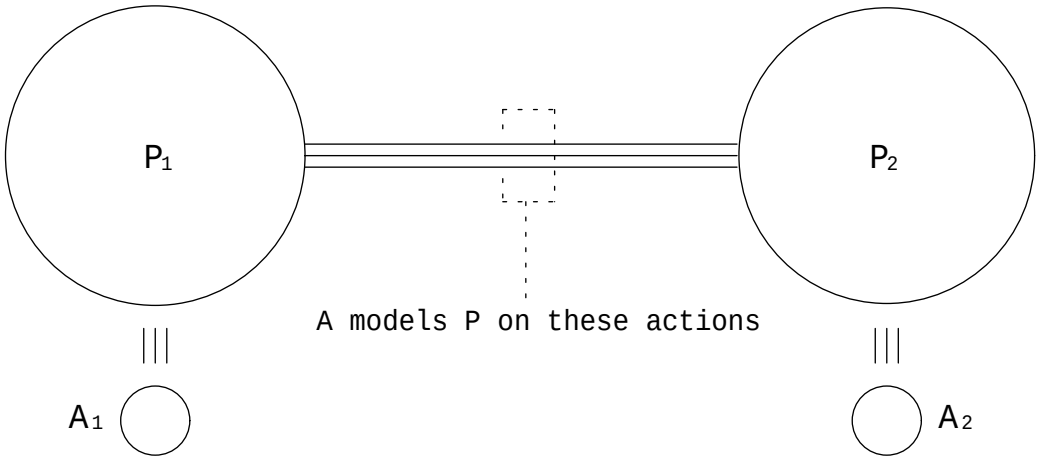

Figure 3: The interface rule

2 The SML programming language

Since the SML language forms the basis of our new compositional language, we give a brief and informal description of it here. A full description is contained in [3, 4]. Other state machine languages are described in [1, 12, 15, 16]. Although SML was developed for specifying complicated finite state machines, it has many of the standard control structures found in modern imperative programming languages, including a while statement, a conditional, a case statment, and a parallel execution statement. There is even a simple mechanism for declaring non-recursive procedures. However, the only data types allowed are booleans and fixed width integers. Thus, any program written in SML has only a finite number of states and can be compiled into a finite state transition table.

All SML programs represent synchronous circuits. At a clock transition, the program examines its input signals and changes its internal state and output signals accordingly. Since we are dealing with digital circuits, the basic data type is boolean. Each boolean variable may be declared to be either (1) an input changed only by the external world but visible to the program, (2) an output changed only by the program but visible to the external world, or (3) an internal variable changed and seen only by the program. Internal non-negative integer variables are also provided but are not discussed in this brief survey of the language.

Since SML programs are intended to be compiled into synchronous finite state machines, the semantics of SML must specify not only what each statement does, but how long the statement takes to execute. These semantics are based on the hardware implementation of a state machine. In such an implementation, combinational logic is used to compute the next state given the current state. At the next clock cycle, this new state becomes the current state and the process repeats. If the clock is slow enough, the combinational logic will always have time to settle. In this case, the state machine will operate exactly the same as an abstract machine in which the next state is computed instantaneously and in which changing state requires waiting for the next clock cycle. When an SML program is compiled

into a finite state machine, the control constructs are what determine the next state; thus they correspond to combinational logic, and are assumed to execute zero time. Assignment statements change the state, and are assumed to take one cycle. This is the basic idea behind the semantics of SML. Exact details on the timing for the individual statements will be described below.

An SML program has the following form:

program ⟨identifier⟩;
⟨declaration list⟩
⟨statement list⟩
endprog

where ⟨identifier⟩ is the name of the program, ⟨declaration list⟩ is a sequence of variable and procedure declarations separated by semicolons, and ⟨statement list⟩ is a sequence of statements separated by semicolons.

Boolean input variables cannot be assigned new values, since inputs are changed by the environment only. Boolean output and boolean internal variables may be changed by:

raise (⟨variable⟩)
lower (⟨variable⟩)
invert (⟨variable⟩)

Each of these statements delays until the next clock transition, at which time the value of ⟨variable⟩ will be changed. The *raise* statement will assert ⟨variable⟩ (make it active), *lower* will negate it, and *invert* will force a change of value.

There are two types of looping statements in SML: the *while* statement and the *loop* statement. The *while* statement has the following syntax:

while ⟨boolean expression⟩ *do loop*
⟨statement⟩
endloop

At the beginning of the *while*, the ⟨boolean expression⟩ is evaluated, and nothing is done (in zero time) if the expression is false. If it is true, ⟨statement⟩ is executed. If ⟨statement⟩ completes execution in no time, the *while* statement delays until the next clock transition and then restarts the loop. If ⟨statement⟩ completes execution after some delay, the *while* statement is immediately restarted. The *exit* statement is used to jump out of the smallest enclosing *while* or *loop* statement. We will not discuss the syntax and semantics of the *loop* statement, since its behavior is similar to the *while*. Neither will we discuss the conditional statement or the *switch* statement, as they are similar to constructs in common imperative programming languages.

The *parallel* statement provides a form of synchronous parallelism. This statement has the form:

parallel
  ⟨statement1⟩ ||
  ⟨statement2⟩ ||
  ...
endparallel

The statements in the *parallel* construct execute concurrently in lockstep. The *parallel* terminates when all of the statements in the *parallel* have finished executing or a *break* is executed. The effect of the *break* statement is to immediately jump out of the smallest enclosing *switch* or *parallel* statement. One of the major uses of the *break* statement is to stop normal processing when an "interrupt" occurs.

In some cases, the timing rules of SML prevent complicated relationships from being simply described without delaying for more than one clock cycle. To alleviate this problem, SML has a statement of the form:

compress ⟨statement⟩ *endcompress*

The effect of the *compress* statement is calculated as if variable assignment takes no time in ⟨statement⟩. Then, after delaying one clock cycle, any changes made by the *compress* statement actually take effect. As an example, consider the following program fragment.

```
compress
  x := (x+y)*(z-w);
  if (x < 5) then
    x := 0;
  endif
endcompress
```

Without the *compress* statement, the first assignment would take one unit of time, and if the condition in the *if* was true, another time step would be required to set `x` to zero. With the *compress*, only one time unit is required in either case.

Although our description of the language has been quite brief, it should be sufficient to understand the example in the next section. The compilation of SML programs in to Moore Machines is described in more detail in [4]. Considerable effort has spent in making the compiler as fast and efficient as possible. The state transition tables produced by the compiler may be implemented in hardware as PALs, PLAs, or ROMs. Various programs have been developed to make this last phase largely automatic. For example, a post-processor is available that produces output which is compatible with the Berkeley VLSI design tools.

3 Compositional SML

The semantics of an SML program is given operationally in terms of a Moore machine, which allows us to apply the CTL model checking algorithms to verify that a program satisfies a formula in CTL. The SML language lacks a notion of parallel composition, however, which satisfies the conditions of the interface rule, and hence is unsuitable for compositional specification and verification techniques. To remedy this situation, we developed a strict extension of SML called CSML (for *compositional* SML). The semantics of a CSML program is given in terms of a collection of Moore machine modules which execute synchronously in parallel. The CSML compiler produces a separate state table for each of these modules. A parallel composition operator for Moore machines gives an equivalent semantics in terms of a single Moore machine. This operator along with the standard definition of Moore machine equivalence satisfy the conditions of the interface rule. This allows us to generate reduced interface processes by hiding appropriate outputs and minimizing using the familiar Hopcroft algorithm [13]. The composition of these reduced machines is produced by a separate program, and is used as input to the CTL model checker. Meanwhile, the modules produced by the compiler can be input to design tools to be translated into various VLSI structures.

Here we describe the two basic extensions to SML which comprise the CSML dialect. The formal definition of Moore machine composition which underlies these constructs is described in [10], along with the proof that it satisfies the interface rule conditions. The CSML construct which corresponds to a Moore machine module is called a process. A *process* in CSML has the following syntax:

process ⟨identifier⟩;
  ⟨declaration list⟩
  ⟨statement list⟩ or ⟨process list⟩
endproc

If the process has an SML statement list as its body, these statements are compiled according to the usual SML semantics into a Moore machine state table. Otherwise, the meaning of the process statement is the parallel composition of its child processes. Variables of the parent process may be referenced by the child processes, provided they are declared as either inputs or outputs in the child process. Our definition of parallel composition requires that no two processes declare the same variable as an output, however. Also, by the definition of parallel composition, variables which are not declared as outputs by any process become external inputs to the program. The compiler generates a unique name for any variable declared internal to a process, based on the path to that process in the process hierarchy. In order to maintain strict upward compatibility with SML, the root process is defined with the *program* keyword.

The other way in which CSML extends SML is the *processtype* statement, which defines a reusable process type. The *processtype* statement may appear in the declaration list of a program or process, and has the following form:

processtype ⟨identifier⟩ (⟨formal parameter list⟩);
  ⟨declaration list⟩
  ⟨statement list⟩ or ⟨process list⟩
endtype

The formal parameters must be declared as inputs or outputs in the declaration list. Process type identifiers which are defined in this way may be referenced only in the lexical scope of the process in which they are defined. A process type is instantiated by a statement of the following form:

process ⟨process identifier⟩ : ⟨processtype identifier⟩ (⟨actual parameter list⟩);

This creates a process by substituting the variables in the actual parameter list for variables in the formal parameter list of the process type declaration. Any other input or output variable names in the process type declaration are resolved in the context of the process type definition. In other words, CSML variables are staticly scoped rather than dynamically scoped. The hierarchical renaming of variables implements this static scoping in a manner similar to the ALGOL 60 copy rule [11].

Figure 4 gives a simple example of a CSML program—a system composed of a producer process and a consumer process which synchronize using a four-phase handshake. A process type is defined in the main program for producer and consumer. The handshake signals exchanged between the two process are defined as internal variables in the main program, and two control outputs `produce` and `consume` are also defined. The producer process waits for the consumer to assert its `request` input. It then pulses `produce` and completes the handshake by asserting `acknowledge`, waiting for `request` to be negated, then negating `acknowledge`. The consumer process asserts `request` then waits for `acknowledge` to be asserted, pulses `consume`, and completes the handshake. Both these process types are instantiated in the body of the main program as `producer1` and `consumer1`.

4 Application: a simple CPU

To illustrate to the use of CSML and compositional methods in designing and verifying controllers, we examine the controller of a simple CPU, with decoupled access and execution units. We define one CSML process to control the access unit, and another process to control the execution unit. We then use the interface rule and the Hopcroft minimization algorithm to produce a reduced interface process to represent the execution unit process. We use this interface process to verify a collection of CTL formulas which comprise the formal specification of the access unit controller.

```
program prodcom;
  output produce,consume;
  internal req,ack;

  processtype Producer(request,acknowledge,produce);
    input request;
    output acknowledge=false,produce=false;
    loop
      while(!request) do loop skip endloop;
      raise(produce); lower(produce);
      raise(acknowledge);
      while(request) do loop skip endloop;
      lower(acknowledge)
    endloop
  endtype

  processtype Consumer(acknowledge,request,consume);
    input acknowledge;
    output request=false,consume=false;
    loop
      raise(request);
      while(!acknowledge) do loop skip endloop;
      raise(consume); lower(consume);
      lower(request);
      while(acknowledge) do loop skip endloop
    endloop
  endtype

  process producer1: Producer(req,ack,produce);
  process consumer1: Consumer(ack,req,consume);
endprog
```

Figure 4: Producer-consumer program

4.1 Architectural description

A block diagram of the CPU is given in figure 5. The CPU is divided into two modules, the access unit (AU) and execution unit (EU), in order to increase its performance by carrying out memory accesses and instruction executions in parallel. The AU's function is to fetch instructions and store them in the instruction queue (IQ), and to maintain a cache of the top location of the stack in a special top-of-stack register (TS). The EU's function is to interpret instructions of the CPU's stack based machine code.

The instruction set has two addressing modes: stack, and immediate, and three basic

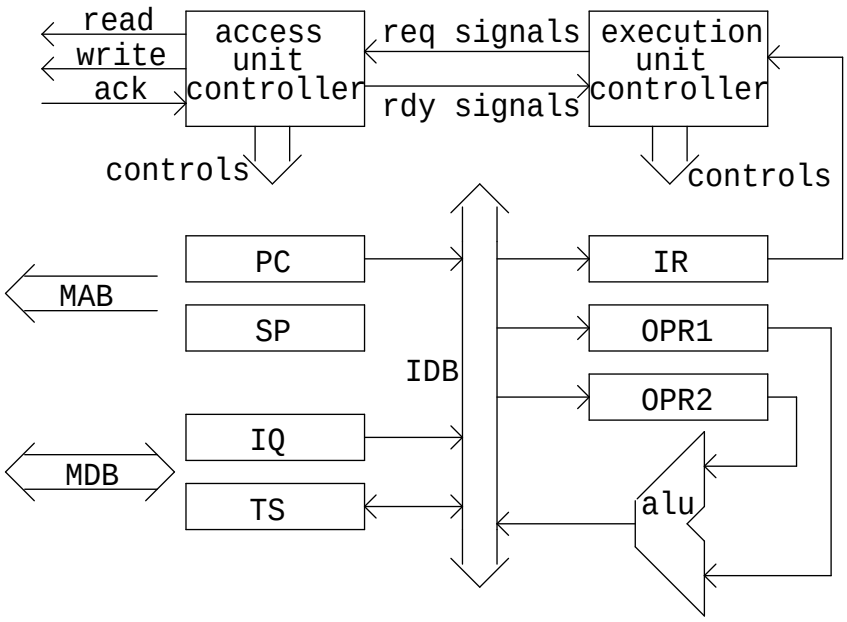

Figure 5: CPU block diagram

classes of instructions: control, one-operand, and two-operand. Instructions that take one operand specify an addressing mode for both source and destination. Instructions that take two operands specify both source addressing modes, and use stack mode implicitly for the destination. The control instructions (branch, call, and return) specify one of eight conditions codes and select either direct or program counter relative addressing.

The access unit has four registers: the program counter (PC), the stack pointer (SP), the instruction queue (IQ), which can hold two instruction words, and top-of-stack register (TS) (see figure 5). The PC is equipped with an incrementer, and the SP with an incrementer/decrementer. The signals generated by the AU controller and their RTL functions are summarized in table 1. These definitions will be of interest later when we discuss the formal specification of the controller.

The execution unit has two operand registers (OPR1 and OPR2), an instruction register (IR), a condition code register (CCR), and an ALU. There is an internal data bus (IDB) by which data are communicated between the EU and AU. The function of the ALU and the signals which control the execution unit data path will not be described here for the sake of brevity.

The access and execute unit controllers communicate via three request signals, *push-req*, *pop-req* and *fetch-req*, three corresponding ready signals, *push-rdy*, *pop-rdy* and *fetch-rdy*, as well as the signal *branch*, which causes the PC to be loaded and the instruction queue to be flushed. The execution unit signals its intention to perform a push, pop or (instruction) fetch operation by asserting the appropriate request signal. If the ready signal is already asserted it proceeds, otherwise it waits for the ready signal to be asserted.

The AU communicates with memory via two busses, the memory data bus (MDB) and the

Signal	Function
fetch	$PC \leftarrow PC + 1 \quad (fetch = fetch\text{-}req \wedge fetch\text{-}rdy)$
PC-MAB	*MAB* (memory address bus) $\leftarrow PC$
PC-IDB	*IDB* (internal data bus) $\leftarrow PC$
branch	$PC \leftarrow IDB$
push	$SP \leftarrow SP - 1 \quad (push = push\text{-}req \wedge push\text{-}rdy)$
pop	$SP \leftarrow SP + 1 \quad (pop = pop\text{-}req \wedge pop\text{-}rdy)$
SP-MAB	$MAB \leftarrow SP$
MDB-IQ	$IQ \leftarrow MDB$
IQ-IDB	$IDB \leftarrow IQ$
TS-MDB	$MDB \leftarrow TS$
TS-IDB	$IDB \leftarrow TS$
MDB-TS	$TS \leftarrow MDB$
IDB-TS	$TS \leftarrow IDB$

Table 1: Access unit control signals

memory address bus (MAB), and via three control signals: *mem-rd*, *mem-wr* and *mem-ack*. The protocol for a memory access is as follows. The AU first asserts one of the memory control signals (*mem-rd* for a read, and *mem-wr* for a write), and causes the appropriate address to be driven onto the MAB (using signals *PC-MAB* or *SP-MAB*). On a write, the AU drives the MDB (using the signal *TS-MDB*). On a read, it loads the MDB data into one of its registers (using signals *MDB-IQ* or *MDB-TS*). It then waits for *mem-ack* to be asserted by the memory system, at which time it completes the access by lowering its control signals.

4.2 Implementing the controllers

In this section, we give an informal specification of the access unit controller and describe some of the CSML code. The AU controller has two functions, which it performs conceptually in parallel: the management of the instruction queue and the management of the top-of-stack cache. We will examine the latter function in some detail. We distinguish three states of the TS register: INVALID, VALID, and MODIFIED. The TS is in the VALID state when its contents match the value in memory pointed to by the SP; it is MODIFIED when the TS has been written, but the contents have not yet been copied back to memory, and it is INVALID otherwise. In particular, the AU is not ready for a push operation when the TS is MODIFIED, because previously pushed data would be lost, and it is not ready for a pop operation when the TS is INVALID, because incorrect data would be read. Figure 6 gives an abstract state diagram which defines the effects of the AU controller operations on the TS register state. This serves as our model of the data path when designing the controller. The CSML code in figure 7 computes the status of the TS and stores it in a variable called

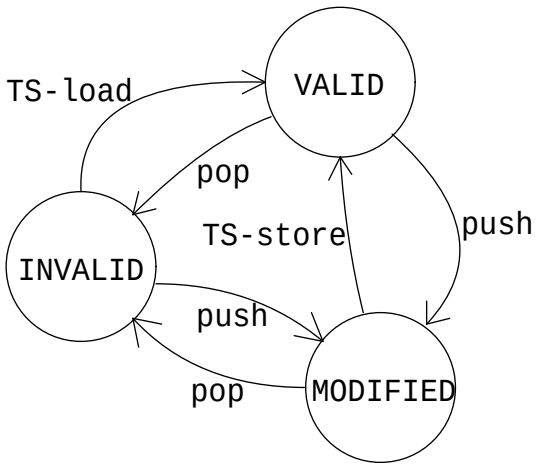

Figure 6: TS manager state diagram

```
loop
  compress
    switch
      case push:
        lower(push-rdy); raise(pop-rdy);
        TS-st := MODIFIED; break;
      case pop:
        lower(pop-rdy); raise(push-rdy);
        TS-st := INVALID; break;
      case TS-load-done:
      case TS-store-done:
        raise(push-rdy); raise(pop-rdy);
        TS-st := VALID; break;
      default: skip;
    endswitch
  endcompress
endloop
```

Figure 7: Code for TS manager

`TS-st`. It also manages the outputs `push-rdy` and `pop-rdy` which signal to the EU that the TS register is ready for a push or pop operation respectively.

In its other capacity as instruction queue manager, the AU controller must simultaneously keep track of the status of the IQ register, fetching a new instruction word when the IQ becomes empty, and flushing the queue when a branch occurs. We will not discuss this function in detail.

Finally, a third parallel thread of control, which acts like a monitor, insures that the TS manager and IQ manager do not attempt to access memory at the same time. The monitor

```
    loop
      switch
        case IQ_st == EMPTY:
          compress read(PC-MAB,MDB-IQ) endcompress;
          break;
*       case TS_st == INVALID & !push-req:
**        compress lower(push-rdy); read(SP-MAB,MDB-TS) endcompress;
          break;
        case TS_st == MODIFIED & !pop-req:
          compress lower(pop-rdy); write(SP-MAB,TS-MDB) endcompress
          break;
        default: skip;
      endswitch
    endloop
```

Figure 8: Code for memory access monitor

```
  procedure read(addrctl,datactl)
          raise(mem-rd); raise(addrctl); raise(datactl);
          while !mem-ack do loop skip endloop;
          lower(mem-rd); lower(addrctl); lower(datactl);
  endproc;
```

Figure 9: Routine `read`.

thread waits in a loop for either the IQ to become EMPTY, or the TS to become MODIFIED or INVALID. It then performs the appropriate memory access: *IQ-load*, *TS-load*, or *TS-store*, respectively. The CSML code appears in figure 8. Note that when the TS register is in the INVALID state, we allow a push request to take priority over a TS-load operation (line `*`), but once the TS-load operation is started, we lower `push-rdy` to prevent push operations from interfering with the memory cycle (line `**`). A corresponding relationship exists between TS-store and pop.

The routine `read` takes as its arguments a control signal to raise to drive the MAB bus, and a control signal to raise to load the IQ or TS registers. It is defined in figure 9. When `read` is called inside a compress statement, only the while loop actually takes time.

The overall structure of the AU controller code is a three-way *parallel* statement as shown in figure 10. The job of the execution unit is more straightforward. It has only one thread of control, and proceeds as follows. It first loads an instruction from the IQ into the IR (i.e., performs a fetch operation). It then decodes the instruction, and jumps to an appropriate routine to interpret that instruction. When the instruction is completed, it starts again. When compiled, the AU and EU controller processes have 13 and 98 states respectively.

```
process AU;
  ... declarations ...
  ... procedures ...
  parallel
  ... memory access monitor ...
  ||
  ... TS manager ...
  ||
  ... IQ manager ...
 endparallel
end proc
```

Figure 10: Overall structure of AU controller code.

4.3 Formal specification for the access unit

In this section we present a formal CTL specification of the access unit controller process. Before proceeding we define a few predicates which will simplify the specifications and the following discussion:

$$\begin{aligned} \textit{push} &\equiv \textit{push-req} \wedge \textit{push-rdy} \\ \textit{pop} &\equiv \textit{pop-req} \wedge \textit{pop-rdy} \end{aligned}$$

$$\begin{aligned} \textit{TS-load} &\equiv \textit{mem-rd} \wedge \textit{SP-MAB} \wedge \textit{MDB-TS} \\ \textit{TS-load-done} &\equiv \textit{TS-load} \wedge \textit{mem-ack} \\ \textit{TS-store} &\equiv \textit{mem-wr} \wedge \textit{SP-MAB} \wedge \textit{TS-MDB} \\ \textit{TS-store-done} &\equiv \textit{TS-store} \wedge \textit{mem-ack} \end{aligned}$$

The predicate *push* indicates that a data word is being pushed onto the stack from the internal data bus. Likewise, *pop* indicates that a data word is being popped off the stack. *TS-load* is true when a memory cycle is in progress which is loading the TS. It indicates that the stack pointer contents are being driven onto the memory address bus (*SP-MAB*), and that the data on the memory data bus are being gated into the TS register (*MDB-TS*). *TS-load-done* is true on the last clock cycle of such a memory cycle (when *mem-ack* is asserted). In a similar fashion, *TS-store* is true when a memory cycle is in progress which is storing the TS value into memory, and *TS-store-done* indicates the last clock cycle of the TS store operation.

The conditions for correct management of the TS manager are derived from the state transition diagram of figure 6. If the TS is in the VALID state, any of the operations *push*, *pop*, *TS-load* and *TS-store* are allowable (the latter two are not present in the diagram, but executing them in this state will cause no harm, since the memory contents match the TS

register). In the MODIFIED state, however, we cannot allow another *push* operation, or a *TS-load* operation to occur before either a *pop* or *TS-store* is completed. This condition is expressed by the following formula:

$$MODIFIED \equiv \forall[\neg(push \vee TS\text{-}load)\mathcal{W}(pop \vee TS\text{-}store\text{-}done)]$$

where $\mathcal{W}$ denotes the *weak until* operator, which is similar to the until operator, but does not require eventual occurrence of its second argument (see section 1). Since the MODIFIED state is entered if and only if a *push* operation occurs, we specify the following formula:

$$\forall G(push \rightarrow \forall X(MODIFIED))$$

In the INVALID state, *pop* or *TS-store* must not occur before either a *push* or *TS-load* is completed. We express this condition in CTL as

$$INVALID \equiv \forall[\neg(pop \vee TS\text{-}store)\mathcal{W}(TS\text{-}load\text{-}done \vee push)]$$

Since the INVALID state is entered if and only if a *pop* operation occurs, we specify the following:

$$\forall G(pop \rightarrow \forall X(INVALID))$$

Of course, we also require that the TS manager not spuriously drive the MAB or MDB busses or overwrite the TS register:

$$\forall G(MDB\text{-}TS \rightarrow TS\text{-}load),$$
$$\forall G(TS\text{-}MDB \rightarrow TS\text{-}store),$$
$$\forall G(SP\text{-}MAB \rightarrow (TS\text{-}load \vee TS\text{-}store)).$$

The first of these, for example, states that the top-of-stack register is loaded from the memory data bus only during a *TS-load* operation.

In order for stack memory cycles to operate correctly, we have the following requirements. First, the address, data and control signals must remain stable during an entire memory cycle. This means that, if a *TS-load* or *TS-store* condition occurs, that condition must persist up to and including the clock cycle when *mem-ack* is asserted by the memory system. Further, as the address must not change during a memory cycle, we require that during *TS-load* and *TS-store* cycles, the stack pointer not change. These requirements are expressed in the following formulas:

$$\forall G(TS\text{-}load \rightarrow \forall((TS\text{-}load \wedge \neg(push \vee pop))\,\mathcal{W}\,TS\text{-}load\text{-}done)),$$
$$\forall G(TS\text{-}store \rightarrow \forall((TS\text{-}store \wedge \neg(push \vee pop))\,\mathcal{W}\,TS\text{-}store\text{-}done)).$$

Six more formulas, which we omit here, define correct management of the instruction queue. The following two formulas state that no spurious memory accesses occur.

$$\forall G(mem\text{-}wr \rightarrow TS\text{-}store),$$
$$\forall G(mem\text{-}rd \rightarrow (TS\text{-}load \vee IQ\text{-}load)).$$

All of the above formulas represent safety properties, i.e., they are characterized by the statement "nothing bad ever happens." Unfortunately, they cannot form a complete specification, since a controller which did nothing at all would satisfy all of the above assertions. Thus, we include the following liveness requirement, which states, in effect, that the CPU always eventually executes another instruction:

$$\forall G \forall F fetch.$$

4.4 Summary of model checking results

Finally, we describe the application of the CTL model checker to automatically verify that our controller meets the above specification. Compiling the CSML code produces a file with state tables for two Moore machines, representing the AU and EU controllers. The AU controller has 13 states, while the EU controller has 98 states. If we computed the parallel composition of the two machines at this stage, the result would have 1274 states. Instead, we apply the interface rule. Because the specification concerns only the inputs and outputs of the AU controller, we can restrict the EU controller to those signals which interface with the AU controller. This means hiding the outputs which control the EU data path. We then minimize the EU controller, obtaining an equivalent *interface process* with only 17 states. Finally, we compute the composition the AU module with this interface process, plus a two state interface process representing the memory system, obtaining a Moore machine with 196 states. This is the machine that we use as input to the CTL model checker to verify the specification.

A sample run of the model checker is depicted in figure 11. The first input to the model checker is a set of fairness constraints, which allow the user to specify which computation paths are considered to be fair executions. In this case, we consider an execution to be fair if the memory system eventually produces an acknowledge signal for every request. The fairness constraint is that infinitely often, either the AU controller is not asserting a request, or the memory system is asserting acknowledge. After reading in the fairness constraints, the model checker is ready to accept macro definitions and CTL formulas to check. For each CTL formula, the model checker determines whether the formula is true or false in the model, and also produces a counterexample for formulas which are false. In fact, in the original version of the controller, there were two bugs in the design which were pointed out by the model checker. The first was that, during a branch, the EU controller did not check to make sure that a *IQ-load* operation was not in progress before modifying the PC. This caused the address on the MAB to change during a memory cycle. The second bug was that the *TS-store* code in the memory access loop incorrectly asserted *MDB-TS* instead of *TS-MDB*. Figure 12 shows the counterexample produced by the model checker which pointed out this error. The total time to verify the 16 formulas of the AU specification on the (corrected and reduced) 196 state model was 36 seconds, running on a Sun-3 workstation.

Using the interface rule, we are able to reduce a 1274 state model of the controller to a 196 state model which is equivalent with respect to all CTL formulas over the inputs and outputs of the AU controller. This amount of reduction is largely due to the nature of the interface between EU and the AU. While the EU interprets a large number of instructions,

```
% mcb -c cpu.fsm
CTL MODEL CHECKER (version B1.0)

Fairness constraint: mem_ack | ~(mem_rd|mem_wr).
Fairness constraint: .

|= push() := push_req & push_rdy.
Macro push defined.

...

|= MODIFIED() := (~(push | TS-load)) AW (pop | TS-store-done).
Macro MODIFIED defined.

|= INVALID() := (~(pop | TS-store)) AW (push | TS-load-done).
Macro INVALID defined.

|= AG (push -> AX MODIFIED).
The formula is TRUE.

|= AG (pop -> AX INVALID).
The formula is TRUE.

...

|= AG AF fetch.
The formula is TRUE.
```

Figure 11: Sample model checking session.

the memory accesses for these instructions fall into a few basic patterns. For this reason, very little of the complexity of the EU is observable via the signals connecting it to the AU, thus the EU controller reduces to a very simple interface process. This might be viewed as a principle of good interface design: that interfaces should reveal as little of the complexity of the underlying modules as possible. Design according to this principle will reduce the global effects of local changes in the design, and simplify the verification process. The interface rule provides a way of quantifying this effect in terms of the number of states in the interface processes.

```
...

|= AG (push -> AX MODIFIED).
The formula is FALSE.
Do you want to specify the input in the initial state? [n]
State 0-0: push_rdy
State 1-4096: fetch_req push_rdy mem_rd MDB_IQ PC_MAB
State 2-0: fetch_req mem_ack push_rdy mem_rd MDB_IQ PC_MAB
State 3-4096: fetch_req push_rdy fetch_rdy
State 5-0: fetch_rdy mem_rd MDB_TS SP_MAB
State 33-4097: pc0 fetch_req fetch_rdy mem_rd MDB_TS SP_MAB
State 62-40: exeu-ir2 exeu-ir4 mem_ack mem_rd MDB_TS SP_MAB
State 94-0: push_req pop_rdy push_rdy
State 121-0: pop_rdy mem_rd MDB_IQ PC_MAB
State 121-4096: pop_rdy mem_rd MDB_IQ PC_MAB
State 120-0: mem_ack pop_rdy mem_rd MDB_IQ PC_MAB
State 123-4096: pop_rdy fetch_rdy
State 135-4096: branch fetch_rdy mem_wr MDB_TS SP_MAB
State 141-0: mem_ack mem_wr MDB_TS SP_MAB
State 39-4096: fetch_req pop_rdy push_rdy
State 71-4096: fetch_req pop_rdy push_rdy mem_rd MDB_IQ PC_MAB
State 113-0: fetch_req mem_ack pop_rdy push_rdy mem_rd MDB_IQ PC_MAB
State 88-4096: fetch_req pop_rdy push_rdy fetch_rdy
State 57-4096: pop_rdy push_rdy fetch_rdy
State 89-4097: pc0 fetch_req pop_rdy push_rdy fetch_rdy
State 60-40: exeu-ir2 exeu-ir4 pop_rdy push_rdy
State 90-0: push_req pop_rdy push_rdy mem_rd MDB_IQ PC_MAB

...
```

Figure 12: A bug found by the model checker

5 Future Directions and Conclusions

We should point out that the task of verifying the CPU does not end with the verification of the controllers. It is necessary, of course, to provide a formal specification of the CPU as a whole, and to prove on the basis of the controller specification and a formal model of the data path circuitry that the CPU specification is valid. The techniques described here are not sufficient to do this in an automated way, because of the very large state space of the data path part of the system. A technique called symbolic model checking, which uses Boolean decision diagrams [7] might be used for this purpose. Another approach to this problem might be to integrate the CTL model checker with an automatic theorem prover (or proof checker), which could perform the final step. We leave the problem of integrating control and data as an open one here, and an area for future research.

Even with the module feature, CSML has some limitations. Perhaps the most difficult issue is how to deal with nondeterminism. Currently, SML processes are completely synchronous and deterministic. In practice, however, it is important to be able to reason about processes that run on different clocks or execute asynchronously. Another important use of nondeterministic processes is to form an abstract representation of a class of deterministic machines. Such a process can be used to prove properties of the entire class, often with greatly reduced complexity [14]. More research is needed to handle this problem within our current framework.

Clearly, the CPU design presented here was not intended to be a practical one. From a practical point of view, however, at least one criticism of CSML should be made. The Moore-machine semantics of CSML (and its predecessor SML) require that raising or lowering a signal always involves one clock cycle of delay. As an example, in the instruction fetch routine of the EU, one clock cycle is simply wasted in order to raise the signal *fetch-req*. This same consideration also made it necessary to use "ready" signals (essentially a pre-acknowledge), since it is not possible to respond to a request with an acknowledge in the same clock cycle. One advantage of the Moore-machine semantics is that all signals between modules effectively pass through a pipeline register. This means that critical path timing of modules can be verified independently. Nonetheless, a language with Mealy machine semantics might be more useful for practical designs.

Finally, additional research is needed on techniques for compositional reasoning about SML processes. The interface rule handles formulas that are boolean combinations of temporal properties of the individual processes. We are currently unable to handle more general properties involving temporal assertions about several processes. Furthermore, in some verification problems it may be necessary to combine the use of the interface rule with proofs of validity for certain CTL formulas. In general, such proofs require a complicated decision procedure. We believe, however, that it will be possible to use the model checker to verify temporal formulas over complex models, which can then be used as lemmas in simple hand-constructed proofs, which might be checked automatically.

As we have seen, the technique of compiling reactive control programs into state transition tables needs not suffer from the state explosion problem, provided compositional techniques are used. Dividing such a system into communicating modules can also reduce the complexity of automatic verification, provided the interfaces between modules hide most of their internal complexity. Although we applied the technique to a hardware example, the same tools can be applied to reactive software systems as well, provided some portion of the system involves finite state control. Driving the system from the state tables produced by the CSML compiler would not only increase the performance of the system, but would also allow some properties of the system to be verified automatically.

References

[1] G. Berry and L. Cosserat. The esterel synchronous programming language and its mathematical semantics. Technical report, Ecole Nationale Superieure des Mines de

Paris, 1984.

[2] M. C. Browne. An improved algorithm for automatic verification of finite state machines using temporal logic. In *Proceedings of the Conference on Logic in Computer Science*, June 1986.

[3] M. C. Browne. *Automatic verification of finite state machines using temporal logic.* PhD thesis, Carnegie Mellon University, 1989.

[4] M. C. Browne and E. M. Clarke. SML: A high level language for the design and verification of finite state machines. In *IFIP WG 10.2 International Working Conference from HDL Descriptions to Guaranteed Correct Circuit Designs, Grenoble, France.* IFIP, September 1986.

[5] M. C. Browne, E. M. Clarke, and D. L. Dill. Automatic circuit verification using temporal logic: Two new examples. In *Formal Aspects of VLSI Design.* Elsevier Science Publishers, 1986.

[6] M. C. Browne, E. M. Clarke, D. L. Dill, and B. Mishra. Automatic verification of sequential circuits using temporal logic. *IEEE Transactions on Computers*, C-35(12), December 1986.

[7] R. E. Bryant. Two papers on a symbolic analyzer for MOS circuits. Technical Report 87-106, Carnegie Mellon University, 1987.

[8] E. M. Clarke and E. A. Emerson. Synthesis of synchronization skeletons for branching time temporal logic. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, 1986.

[9] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, 1986.

[10] E. M. Clarke, D. E. Long, and K. L. McMillan. Compositional model checking. In *Proceedings of Fourth Symposium on Logic in Computer Science*, 1989. To appear.

[11] P. Naur (*ed.*). Revised report on the algorithmic language ALGOL 60. *Comm. ACM*, 6(1):1–20, 1963.

[12] D. Harel. Statecharts: A visual approach to complex systems. Technical Report CS84-05, The Weizmann Institute of Science, February 1984.

[13] J. E. Hopcroft. An $n \log n$ algorithm for minimizing the states in a finite automaton. In *The Theory of Machines and Computation*, pages 189–196. Academic Press, New York, N.Y., 1971.

[14] R. P. Kurshan. Reducibility in analysis of coordination. In *LNCIS*, volume 103, pages 19–39. Springer-Verlag, 1987.

[15] D. L. Parnas. A language for describing the functions of synchronous systems. *Communications of the Association of Computing Machinery*, 9:72–75, February 1966.

[16] J. D. Ullman. *Computational Aspects of VLSI.* Computer Science Press, 1984.