Symbolic Analysis and Verification of CPA Descriptions

M. C. McFarland, S.J.
Boston College
Chestnut Hill, MA 02167

T. J. Kowalski
AT&T Bell Laboratories
Murray Hill, NJ 07974-2070

ABSTRACT

CPA is a formalism for specifying the behavior of digital systems. It describes the input/output behavior, independent of the internal structure and operation of the system. The primary use for CPA is formal verification of digital designs, in which the behavior of a design is checked for consistency against the CPA specification. This paper describes `audit`, a system we are developing that formally verifies a digital design against a CPA specification. The design is represented as a state machine with multi-bit data registers that can be tested and changed on the state transitions. Whereas many verification programs that work at this level expand the data part of the machine into individual states, `audit` treats the registers as symbolic entities and uses symbolic simulation and first-order predicate calculus to reason about their effect on the behavior of the machine. Yet unlike other symbolic simulation programs, `audit` can automatically analyze the behavior of simple loops and can reason about *sequences* of inputs and outputs.

1. INTRODUCTION

1.1 Formal Verification with `Audit`

CPA is a formalism for specifying and reasoning about the behavior of a digital system at the register-transfer and finite state machine (FSM) level. CPA uses an abstract, nonprocedural style of specification. It describes the input/output behavior of a system using the sequences of input and output events at its ports. It is therefore independent of the internal structure and operation of the system. This makes it a good medium for formal specification and verification, because it gives a view of the system that is orthogonal to that of the implementation. Conceptual errors in the implementation are less likely to be duplicated in the specification and are therefore more likely to be exposed.

This paper describes a program called `audit` that we are developing to check a

digital system design against a CPA specification to determine whether the properties given in the specification hold for the design. The design is represented as an *extended* finite state machine (EFSM), that is, a state machine with multi-bit registers added to it, where the conditional transitions can depend on the values of the registers, and the transitions can do arithmetic on the registers. Thus the machine has both a data state, which is the value of the registers, and a control state; these work together to determine the behavior of the machine. Any verification procedure for EFSMs must take this into account. Most verification systems expand the data registers and merge them into the control so that each distinct set of values for the data registers is represented as a separate control state. `Audit`, however, treats the elements of the data part as separate entities and reasons symbolically about their values and their effect on the control. Yet unlike previous verification programs that dealt with a symbolic data state, `audit` can automatically analyze many types of loops, and it can reason about sequences of inputs and outputs, not just a simple mapping from initial inputs to final outputs.

1.2 Relation to Previous Work

CPA is similar to temporal logic[1,2] in that both can express relations between progressive states of a system. Both are therefore good for expressing reachability and other liveness properties. However, temporal logic is not as effective for expressing functional relations or timing constraints, and it is only with great difficulty, if at all, that temporal logic can express the relations between indexed members of a sequence. Nevertheless, we learned a great deal from the work on model checking by Clarke *et al.*[2]

Symbolic simulation was one early technique used in the formal verification of hardware.[3,4] Those early systems, and later ones, worked on acyclic program or state graphs, and they found the mappings from initial inputs to final outputs. They did not deal with sequences of inputs and outputs. Hunt[5] and others have used a form of symbolic simulation and the Boyer-Moore theorem prover, with its ability to do inductive proofs, to verify recursive or iterative hardware descriptions, but with a good deal of intervention and coaxing by the user.

Recently most work in the automated verification of sequential hardware has involved the comparison of finite state machines[6] or ω-regular languages.[7] These techniques have proved powerful because they operate in the Boolean domain, where the relations are simple and well-defined and the problems are decidable. The problem is that when a machine has a significant data part, all its data states must be translated into separate control states of the finite state machine being analyzed. If there is one register with n bits, that register has 2^n states. When that register is included with a machine M, the number of states in the combined machine is the number of states in M times 2^n. Several brilliant techniques have been used to slow this explosive growth in the number of states, including the implicit enumeration of the states in a state graph[8] and the use of ordered binary decision diagrams[9] to represent the implicit

states.[10] But these do not eliminate the problem. Keeping the data state separate, as `audit` does, requires a more complex form of analysis; but the size of the problem is independent of the size of the registers.

The HOL system[11] provides a general environment for hardware verification. Because all signals can be described as functions, it can certainly describe sequences and express timing and other constraints. However, HOL is too general a formalism to allow automated proofs, although sequences of steps in a proof can be automated.

There has been limited progress in automating the analysis of loops. On the software side, Wegbreit has suggested some rules for synthesizing inductive assertions,[12] but they are not very robust and are difficult to automate. For hardware, where many loops tend to be simple, often involving little more than counting, Devadas, Keutzer and Krishnakumar have developed a method using symbolic analysis of the data state to do a reachability analysis of simple loops where the variables are only set to a constant or incremented.[13] Yannakakis and Lee have proposed a method for performing a reachability analysis of efsm's whose variables undergo separable affine transformations.[14] A separable affine transformation is one in which each variable v is assigned a value of the form a*v+b, where a and b are constants. The analysis can be extended to the case where v is assigned a value a*u+b, where u and v are mutually dependent, but not dependent on any other variables. In either case the analysis involves solving a linear programming problem of tractable size.

Those systems that analyze efsm's by reducing them to fsm's and generating the entire state space, or the reachable part of it, generally handle loops by "executing" the transitions in them repeatedly as long as they keep leading to new states. In this way they eventually generate all states reachable by any number of iterations.[15]

The algorithm we have developed for `Audit` handles loops of a more general class of loops than either Keutzer, Devadas and Krishnakumar or Yannakakis and Lee. It treats the data state symbolically and only goes through the loop once, so it is not dependent on the size of the variables. And it gives a complete characterization of the behavior, not just the set of reachable states, so it can be used to analyze or verify a broader class of properties than most efsm verification systems.

1.3 Outline

In the following section we will give a brief overview of CPA, the formalism we use to specify the properties of a system that need to be verified. Then we will describe the basic verification algorithm and explain how it works. Next we will show how `Audit` analyzes loops. This will be followed by two simple examples to illustrate how the system works. Finally, we will describe the current status of the system, and note what we have learned from it so far.

2. CPA

CPA developed from a study of several behavioral specifications that were used in system design.[16] These specifications, we found, focused on input/output behavior, not on the internal structure or operation of the system being specified. Therefore they described behavior by input/output *events*, that is, the reception and transmission of values and signals through the system's input and output connections. A system specification generally consisted of a set of properties that determined what sequences of events were required or allowed for the system to operate correctly.

In analyzing these specifications, we identified four different types of properties:

- **Precedence Relations.** These relations specify the order in which inputs and outputs must occur. They are typically shown by timing diagrams. For example, part of the specification of a bus interface might be that when it receives a *Ready* signal, it must place the data on the *Data* bus and set the *Acknowledge* signal.
- **Functional Relations.** These relations show how the values input and output by the system are related. They show the computations performed by the system.
- **Execution Constraints.** Execution constraints are bounds (both upper and lower) on the number of times certain events can occur and on the delay between events. Maximum and minimum timing constraints fall in this category.
- **History.** Precedence relations, functional relations and execution constraints often involved whole sequences of events, not just individual events. For example, the parity bit of a transmitter might depend on the values of the last seven bits sent.

CPA is a formalism for behavioral specification that integrates all four classes of properties mentioned above. It is based on events and precedence relations, so requirements on the ordering of events are easy to express. It supports constraints on events, which allows the description of functional and timing relationships. Finally, in CPA events are indexed, which gives a straightforward way of describing event sequences and therefore history.

CPA has several other characteristics that we found to be important for developing specifications. It gives an abstract view of behavior, so that it is possible to specify the system without determining its internal design. It allows partial specification and incremental development of specifications, because a system is described by individual properties. It is conceptually simple, based on just a few primitives. And it supports hierarchical descriptions, so behavior can be described at several different levels, with well-defined relations between those levels.

2.1 CPA's Model of Behavior

CPA models a system as one or more processes, where each process is described at the extended state machine level. This means that each process interacts with its environment, including other processes, through a series of discrete events, each representing

the transmission or reception of some digitally coded information. Thus the model does not describe the electrical properties of systems, but it does include real-time behavior.

Each process in a CPA description has a set of ports defined for it, where each port is either an input or an output and has a specific type. The type gives the bit width of the port, or equivalently the range of values it can take. For example, a process named *sender* with two ports, an eight-bit output port *data* and a one-bit input port *ack*, would be declared in CPA as:

process sender(out data<7:0>, in ack<0>){}

A process interacts with the external environment and other processes by reading and writing values through its ports. All behavior is described through these reads and writes. Regarding internal structure and operations, each process is a black box.

A process has control of its reads as well as its writes. In other words, a process reads an input when it decides to, not necessarily when the input appears. A system specification, however, will usually contain constraints that determine how quickly a process must respond to externally supplied inputs.

The act of reading or writing a value on a port is called an *event*. A fully determined event has three components:

The *port* N where the event occurs
The *value* V read or written
The *time* T when the event occurs

An event E can therefore be written as a triple <N, V, T>. For any such event E, we use selectors .n, .v, and .t to identify the individual components. In other words, if E = $<N_0, V_0, T_0>$, E.n = N_0, the port name for E, E.v = V_0, the value for E, and E.t = T_0, the time of E.

2.2 Specification in CPA

Within this model, specifying the behavior of a system means determining the acceptable sequences of input and output events. We do this in CPA by writing a series of expressions called *event expressions*. Each event expression has three parts:

- A **Condition**: a logical formula giving the conditions under which this expression applies;
- A **Precedence Relation**: a required ordering between two or more events;
- An **Assertion**: a logical formula specifying constraints on the sequence numbers, values, and times of the events named in the precedence relation.

It is from these three components that the formalism gets its name.

In our implementation of CPA, the syntax of an elementary event expression is

when C {Θ_1 --> Θ_2} where A;

C and A are the condition and assertion, respectively. Both are formulas in first-order predicate calculus over events, their values, and their times. They are written using

the following keyboard-friendly logic notation: !x. P(x) means that there exists some x such that P(x) holds; @x Q(x). P(x) means that P(x) holds for all x such that Q(x) is true; && stands for AND; || for OR; ~ for NOT; and → for IMPLIES.

Θ_1 --> Θ_2 is the precedence relation. It says that whenever an event of type Θ_1 occurs, it must be followed by an event of type Θ_2. Θ_1 and Θ_2 are called *event schemas*. In event schemas, events are identified primarily by the ports on which they occur. Thus, for example "sender::data" stands for an event that outputs a value on the *data* port of process *sender*. An event name can be qualified by adding a constraint on the value read or written by that event. "Sender::data(.v != 0)," for example, stands for an event that writes a nonzero value on port *data*, while "sender::ack(.v == 1)" stands for an event that reads a 1 on port *ack*.

A particular event E *matches* an event schema N(P) if E.n == N and P(E.v) is true. For example, any event of the form <sender::ack, 1, T> matches the schema "sender::ack(.v == 1)," while an event of the form <sender::ack, 0, T> does not. Neither does an event of the form <sender::data, V, T>, of course. If the constraint P is missing, it is taken to be true, i.e., all values satisfy it.

The reason for having the constraint as part of the event schema rather than in the condition or assertion is that it affects the way events are counted. For example if u is an output variable in process P, the event schema "P:u(.v==1)" refers only to events that output a 1 on u. Other events affecting u are ignored. This is significant when events are *indexed*, as explained below.

Indexing further distinguishes among the events that match a given event schema. The time component of events defines a natural ordering on them:

$E_1 < E_2$ if and only if $E_1.t < E_2.t$

Using this order, any set of events $\{E_i\}$ can be organized into a sequence E_1, E_2, $\cdots$, where $E_j < E_{j+1}$ for all j such that E_j and E_{j+1} are in the sequence. Then any event in a set is identified by its index—that is, its position—in the sequence derived by ordering the set in time. In our notation we put the index in square brackets. Sender::data[5], for example, is the fifth output from port *data* of process *sender*; and sender::ack[j](.v == 1) represents the jth time a 1 is read on port *ack*. Because we are interested in infinite, repetitive sequences of events, where absolute indices are not usually significant, we will often use *relative indexing* of events. Putting a + or − before an index expression means that index is relative to a reference event X. In an event expression "when C {Θ_1 --> Θ_2} where A;", the event matching Θ_1 is the reference event. All relative indices anywhere in C, A, and Θ_2 implicitly reference it. For example, in the event expression

```
when (sender::ack[−1].v == 0) {sender::ack(.v == 1)  −−>  sender::data[+1]}
        where (sender::data[+1].v == x);
```

sender::data[+1] is the first output at the *data* port following the input of 1 at the *ack* port. Sender::ack[−1].v == 0 refers to the last time *ack* was read before reading a 1 on it. Therefore, the previous event expression can be translated as follows:

If the last time *sender* checked *ack*, it saw a 0 and now it sees a 1,
then its next output on *data* must be *x*.

If the reference event Θ_1 is referred to in C or A and it does not have an absolute index, it is referenced as $\Theta_1[0]$.

For brevity we will sometimes omit the process part of a port name when it is clearly understood. Thus, the assertion in the previous example could also be written data[+1].v == x.

For synchronous processes, we also allow time to be expressed in clock cycles. E.c stands for the cycle in which the event E occurs. We could, for example, state the constraint that each output on B occur on the cycle following the corresponding input on A as B[i].c – A[i].c == 1.

The conditions in event expressions are useful for specifying conditional actions. For example, a circuit M that reads from input port S and writes a one to port A0 if S is zero and a one to port A1 if S is one can be specified by

```
process M(in S, out A0, out A1)
{
        when S[0].v == 0 {S  ==> A0[+1]} where A0[+1].v == 1;
        when S[0].v == 1 {S  ==> A1[+1]} where A1[+1].v == 1;
}
```

More complex conditions can also be expressed. Consider, for example, an interface circuit B, which transfers data from its input *Din* to its output *Dout*. After each output, it waits for an acknowledgement on line *Ack* from the receiver before it sends the next output. If the acknowledgement is not received within a certain time, say 200 clock cycles, B raises a *Timeout* flag. The specification for this behavior is:

```
process B(in Din, out Dout, in Ack, out Timeout)
{
out_eq_in:
        when TRUE {Din  ==> Dout[+1]} where Dout[+1].v == Din[0].v;
check_ack:
        when TRUE {Dout  ==> Ack[+1]}
                where Ack[+1].c – Dout[0].c <= 10;
next_output:
        when (Ack[0].c – Dout[–1].c < 200) {Ack(.v == 1)  ==> Dout[+1]};
check_again:
        when TRUE {Ack(.v == 0)  ==> Ack[+1]}
                where Ack[+1].c – Ack[0].c <= 10;
not_next_output:
        when Ack[+1](.v == 1).c – Dout[0].c >= 200 {
                Dout  ==> Timeout[+1](.v == 1)
        } where Timeout[+1].c – Dout[0].c == 200;
}
```

The first three lines describe a normal transfer. Line out_eq_in states that the value of the output is always the last value read at the input. Line check_ack requires that after every output, the system check the Ack line. This should be done within 10 clock cycles. If a 1 is seen on Ack within 200 cycles of the last output, according to line next_output, the system proceeds with the next output. Line check_again states that whenever a 0 is seen on Ack, the system should check again within 10 cycles. Finally, line not_next_output gives the alternative behavior if the conditions in line next_output are not met. If an acknowledge is not read within 200 clock cycles of the last output, the *Timeout* flag is raised. This is constrained to take place exactly 200 cycles after the last output.

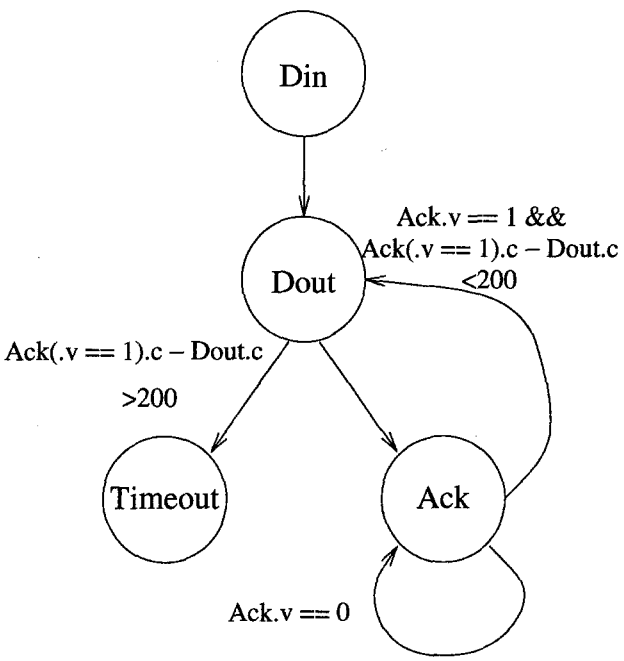

Figure 1. Event Graph for a CPA Specification

Each event expression can be visualized as an arc in an event graph,[17] where each node in the graph is an input/output event, and each arc is a path the system can take from one event to the next. Arcs can be labeled by the conditions under which they can be traversed (C) and constraints on how they are traversed and what their effects are (A). Figure 1 shows how the specification fragment for the timeout example given previously can be interpreted as an event graph. The paths in the graph are annotated with the conditions under which they are taken, but the assertions are not shown.

2.3 Semantics

We have developed a formal semantics for CPA, defining an event expression by the sequences of events that satisfy it.[18, 19] Informally the definition is as follows. An event expression "when C {Θ_1 –> Θ_2} where A;" is satisfied by a machine M if the behavior of M is such that whenever an event matching Θ_1 occurs and C is true, it must be followed by an event matching Θ_2 and A must be true of these events.

3. AUDIT

3.1 Overview

The inputs to `audit` are a CPA specification and an EFSM machine description. The CPA specification consists of a process definition and a set of event expressions describing the behavior required of the process. The EFSM description is similar to a FSM description, consisting of a set of inputs, a set of outputs, a set of (control) states and a set of transitions. An EFSM may also have a set of registers, meaning internal memory elements, defined for it. The inputs, outputs and registers of an EFSM may have sizes larger than one bit. Each transition is defined by its initial state, its final state, the condition under which the transition is taken, and the actions executed when the transition is taken. The condition is a predicate on the inputs and the registers. The action may include assignments to registers and to outputs, where the value assigned may be the result of an arithmetic or logical expression. An example of an EFSM transition is

S2 → S3 when c > 0 {c = c + 1; Y = 1}

where c is a register and Y is an output port. This describes a transition from state S2 to state S3 that is enabled when the value of the register c is greater than 0. When the transition is taken, c is incremented and a one is output on Y.

To verify a CPA description on an EFSM, we must show that each event expression is satisfied by the machine. Given the semantic definition of CPA, the basic procedure for verifying an event expression "when C {Θ_1 –> Θ_2} where A;" on a machine M begins by matching the *reference* event Θ_1, i.e., finding all the transitions where it may occur. Then for each such transition, it must determine the state of the machine at that point, including what is known about the values of the variables. After adding the condition C to the state, it must search forward along each execution path to find a transition matching the *target* event and check whether the assertion A holds at that point.

3.2 Symbolic Simulation

The procedure must take into account the data state of the machine, meaning the values of the registers, and how this affects the machine's execution. Furthermore, because CPA can describe *sequences* of events, the procedure must keep track of

sequences of inputs and outputs that occur when the machine is executing. It does this with a technique called *symbolic simulation*. Symbolic simulation "executes" a machine description, not by keeping track, of the values of registers and outputs for a specific set of inputs, but by keeping track of what is known *about* the register values and outputs for all valid inputs. To do that it maintains a symbolic "state," which is a predicate describing what is true about the register and output values; and as it "executes" each transition, it changes the state to reflect both the condition assumed by the transition and the actions taken on it.

`Audit`'s symbolic state, called the *path state*, represents what is known about the data state of the system and its past execution after a certain series of transitions, called a path, have been executed. Thus a path state is associated not only with a control state, but also with the transitions executed in reaching that state.

A path state has three elements. The first is the set of conditions known to hold at that point in the execution. This is a set of first-order formulas on previous input events and the initial values of registers. For example, at some point in the computation, it might be known that the first value read on input port B was 1 because a transition tested for that. In that case the condition part of the path-state would include the condition B[1] == 1.

The second element in a path state is a list giving the symbolic value of each relevant register in the machine and of certain other important quantities. This is kept as a set of name:value pairs called *bindings*. For example, if register c was known to be 0, the pair c:0 would be in the binding list, while if register v was known to be one more than the second element input at port Y, the binding v:Y[2]+1 would be in the list. The information held in the bindings could be included in the conditions, but because it is used in a special way, it is kept separate.

The final element in a path state is a list of input and output events that occurred in reaching that state. This is needed because the assertion in the event expression to be verified usually includes references to past events. If there are timing constraints to be verified, the entry for an event includes an assertion about the time when the event occurred. For an output event, there is also information about the value output. When an input or output statement is encountered during the search of a path, the entry for the I/O event is constructed using information from the binding list, which keeps track of event indices and time, as well as the values of variables. Suppose, for example, that a transition includes the action "Z = a", where Z is an output port and a is known to have the value 1. Suppose too that Z has been referenced once before on this path. Then Z[2].v == 1 will be added to the event list. Furthermore, if timing information is needed and the elapsed time on this path is 6 clock cycles, then Z[2].c == 6 would also be added.

When the path state at the initial state S1 of a transition S1→S2 is known, the transition can be "executed." This means transforming the path state as required by the

transition, and posting the result at S2. If there is a condition on the transition, that condition is added to the condition list of the path state. If a register v is assigned a new value in the action part of the transition, the entry for v in the binding list is updated with the new value. If there is an output in the action list, an entry for that output is added to the event list. An entry in the event list is also added for any input referenced in the condition or action parts of the transition. Whenever a register is referenced in either the condition or action part, its value is found from the binding list and substituted for the variable.

To find the path state for the path from control state S0 to control state Sn, the procedure starts with a path state containing the initial conditions at S0, and pushes the path state through each transition on the path until Sn is reached. To explore all paths from S0, it starts with the path state at S0, but when a state is reached that has more than one outgoing transition, a copy of the path state is made for each transition and pushed through that transition. When different paths reconverge at a control state, the path states from those paths are joined together in a list of path states called a state vector.

3.3 The Basic Algorithm

Figure 2 outlines the basic verification algorithm used in `audit`. The procedure starts by finding all transitions t:S1→S2 that match E1. It maintains a state vector SV of the path states that hold at S2 including C from the event expression. It searches the paths from S2 using a depth-first algorithm, which is illustrated in Figure 3. It finds all transitions t':S1'→S2' that match E2 and composes a new path state vector SV'. A verification is successful if the path is reachable, the transition t' exists, and the path state in SV' implies the assertion in the event expression.

3.4 Further Considerations

The verification algorithm, as presented thus far, has been highly oversimplified. There are other issues that must be dealt with to do the verification automatically. We mention some of the most important ones here.

3.4.1 Matching event schemas. It is not always obvious when an event encountered in a transition of an EFSM matches the event schema in a CPA description. Both the index and the condition can cause problems. For example, consider the event schema A(.v < 1)[+1], where A is an output port. The candidate transitions for a match are those that output to port A, of course. The +1 shows the first matching event after the reference event. However, the condition .v < 1 means that a transition that writes to A is a match the schema only if it outputs a value less than one. The symbolic simulation may provide enough information to decide if the condition is true, but not necessarily. The value output might depend on values input previously, which are not determined by the machine being analyzed. If the analysis cannot determine whether the condition is satisfied or not, it must do the next analysis twice, once assuming the

```
verify(EE, M)
 /* EE is an event expression C {E1  -->  E2} A
   M is an EFSM */
{
        good = TRUE;
        for all transitions t:S1→S2 such that an event matching E1 occurs in t {
                find SV, the list of path states that hold at S2;
                add C to each path state in SV;
                for all paths starting at S2 {
                        if the path is not reachable (its condition is false)
                                continue;
                        find a transition t':S1'→S2' such that an event
                                matching E2 occurs on t';
                        if no such transition exists {
                                report a failure on this path;
                                good = FALSE;
                        }
                        find SV', the set of path states at S2';
                        for each path state PS' in SV';
                                if !(PS'→ A) {
                                        report a failure on this path;
                                        good = FALSE;
                                }
                        }
                }
        }
        if (good)
                report EE is verified on M;
}
```

Figure 2. Audit's Verification Algorithm

condition is satisfied, and again assuming it is not.

3.4.2 Locality. For an EFSM of realistic size, there are many paths; and the data transformations performed by the full machine can be complex. Nevertheless, most of the properties described by individual event expressions cover only a part of the total state transition graph. The key to keeping the analysis tractable, therefore, is to search only that part of the graph that is required for verifying the event expression at hand. `Audit` tries to do this in several ways. First, in initializing the path states before starting the search from the reference event, it decides what variable values and past events will be needed and looks back in the state transition graph only far enough to pick up what it needs. Second, in compiling its event lists, it only looks at events that are mentioned in the event expression. Finally, as seen above, the search forward from the reference event stops as soon as the target event is found.

```
dfs(EE, Si, SV)
 /*EE is the event expression C {E1  -->  E2} A
   Si is a control state of the machine M
   SV is the list of path-states that hold at Si */
{
        if Si has no out transitions or Si is marked as visited {
                report a failure on this path;
                return FALSE;
        }
        mark Si as visited;
        good = TRUE;
        for all transitions t':Si→Sj out of Si {
                SV' = SV with each path state transformed by ''executing'' t';
                if there is an event in t' that matches E2 {
                        for all path states PS' in SV'
                                if !(PS'→A) {
                                        good = FALSE;
                                        report a failure on this path;
                                }
                } else {
                        result = dfs(EE, Sj, SV');
                        good = good && result;
                }
        }
        return good;
}
```

Figure 3. Depth-First Search Through the State Transitions Graph

3.4.3 Arithmetic reasoning. To know what paths are executable and to determine whether the path state for a certain path implies the assertion in an event expression, `audit` must be able to reason about arithmetic expressions and about sequences. It does this in two ways. First, it has its own simplifier for on-the-fly reduction and simplification. This is modeled on the original program verifier of King,[20] as refined by Sarkar and De Sarkar,[21] with some features added to handle operators and expressions peculiar to `audit`. This internal simplifier can handle most arithmetic and logical reductions, including the associative, commutative and distributive laws, constant-folding, inverses, logical and arithmetic identities, and so on. This allows `audit` to reduce and combine parts of the path states as it goes. Second, for more complicated problems, `audit` translates the facts it knows and the assertions to be proved into the proper form and sends them to Otter, a general-purpose resolution-based automated reasoning program.[22] Otter is needed, for example, for the renaming of quantified variables and functions, for the reduction of complicated logical expressions, and for the substitution of one arithmetic or logical identity into another.

4. LOOP ANALYSIS IN AUDIT

Symbolic simulation works well on acyclic paths, but runs into trouble when there are loops. It is straightforward to find the effect of executing a loop once. The problem is in finding a general characterization of the effect of executing the loop an arbitrary number of times. If the number of iterations were known to be some specific constant k, the loop could be executed symbolically k times. But usually k is not specified or is data dependent, so symbolic simulation by itself does not work. There has to be a way of generalizing its results.

Most symbolic simulation systems have avoided the problem by requiring the user to break all loops in the machine description and to specify and verify the behavior of each one separately. Typically this is done by having the user supply a logical formula *I*, called a loop invariant or inductive assertion, such that (1) if *I* holds at the beginning of the loop, *I* still holds after the loop has been executed once; (2) *I* is implied by the initial conditions of the loop; and (3) *I* implies the desired exit conditions of the loop. If an *I* can be found such that (1)-(3) can be proved to hold, that is enough to guarantee that, given the initial conditions, the desired exit conditions will hold no matter how often the loop is executed, because *I* holds for any number of iterations.

`Audit`, on the other hand, does loop analysis automatically for a class of loops that includes the kind that are most often encountered in hardware. It does this by symbolically executing the loop once, collecting all the conditions, variable values, and events, simplifying wherever possible, and generalizing these for k iterations, using a general solution for the recurrence equation describing the loop.

4.1 Problem Statement

Let M be an efsm with states $\{S_1, S_2, ..., S_p\}$. Suppose M contains a loop L with an initial state S_L, called the head of L. L consists of one or more paths $S_L \rightarrow S_{i_1} \rightarrow S_{i_2} \rightarrow \cdots \rightarrow S_{i_{k-1}} \rightarrow S_L$, where each $S_{i_j} \rightarrow S_{i_{j+1}}$ is a transition in M, as are $S_L \rightarrow S_{i_1}$ and $S_{i_{k-1}} \rightarrow S_L$. Any cyclic path that contains S_L must be considered part of L. If there are any nested loops contained in L, it is assumed that they have already been reduced using this procedure.

Let $\bar{v} = v_1, v_2, \cdots, v_n$ be the registers in M. Then by symbolic simulation we can find the transformations on the registers $\bar{v}$ performed by one pass through the loop L. We use f_j to denote the transformation on v_j. Thus we can write

$$v_j^i = f_j(\bar{v}^{i-1}) \quad \text{for } j = 1 \text{ to } n \tag{1}$$

where v_j^i is the value of v_j after the ith iteration of L and $\bar{v}^{i-1}$ is the vector of values for the registers $\bar{v}$ at the beginning of the ith iteration. f_j can depend on the data input on the ith iteration as well as the initial values of the registers. If there are several

paths in L, f_j could be conditional. The problem is to find a generalization of f_j, $\hat{f}_j$, such that,

$$v_j^m = \hat{f}_j(m, \bar{v}^0) \quad \text{for } j = 1 \text{ to } n \tag{2}$$

where $\bar{v}^0$ is the initial values of the registers in $\bar{v}$ before L is executed and v_j^m is the value of v_j after m iterations of L.

In general, this problem is unsolvable. Under certain restrictions, however, we can find a closed solution.

4.2 Semi-separable Affine Transformations

Define the following order on the registers in $\bar{v}$:

$v_j \leq v_k$ iff $f_k(\bar{v})$ is independent of v_j

In other words, the computation of the new value of v_k does not depend on the value of v_j. Then the registers in $\bar{v}$ are *semi-separable* if and only if there is a total order of the registers in $\bar{v}$ under the $\leq$ relation. That means that we can put all the registers in $\bar{v}$ in order,

$$v_{i_1} \leq v_{i_2} \leq \cdots \leq v_{i_n}$$

such that v_{i_n} is independent of all the other variables, $v_{i_{n-1}}$ is independent of all but v_{i_n} and itself, and so on.

The significance of the semi-separable property is that if the $\hat{f}_j$ exist, finding them reduces to the problem of finding solutions to the recurrence equations

$$v_j^i = f_j(i, \bar{v}^0, v_j^{i-1}) \quad \text{for } j = 1 \text{ to } n \tag{3}$$

The equation for v_{i_n} is already in this form. Once we solve it, we can substitute $\hat{f}_{i_n}(i, \bar{v}^0)$ into equation (1) for $j = i_{n-1}$, reducing it to the form of (3). The solution of this equation could then be substituted into the equation for $j = i_{n-2}$, and so on, until the equations for all the registers have been solved. Note that the function f_j can depend on the iteration number as well as the initial values of the variables. The dependence on the iteration can come both from the use of values of other variables and from the input data read on that iteration.

To find a solution to equation (3), we further restrict the form of the f_j to what we call *generalized affine* transformations. f_j is a generalized affine transformation if equation (3) has the form

$$v_j^i = g(i) * v_j^{i-1} + h(i) \tag{4}$$

where * and + are two associative, commutative binary operations such that * is distributive over +. Normally, for arithmetic operations, * would stand for multiplication

and + for addition, but they could also stand for logical AND and OR respectively. g(i) and h(i) may depend on the iteration, but do not depend on v_j or any of the other registers in $\bar{v}$. They may also depend on the initial values of the registers, $\bar{v}^0$, but, because those are already known, symbolically at least, before the loop analysis, they can be treated as constants. Therefore we do not write the dependence explicitly. The dependence of g and h on the iteration, along with the ability to describe transformations on sequences of inputs and outputs, are what make this method more general than that of Yannakakis and Lee.

The recurrence equation (4) has the following solution:

$$v_j^m = v_j^0 * \prod_{i=1}^{m} g(i) + \sum_{i=1}^{m} \left(\left(\prod_{k=i+1}^{m} g(k) \right) * h(i) \right) \tag{5}$$

where $\prod a_i = a_1 * a_2 * \cdots$ and $\sum a_i = a_1 + a_2 + \cdots$

Theorem. Equation (5) is the solution to the recurrence in equation (4).
Proof. The proof is by induction on m.
Base Case: m = 1. With m = 1, the first product and the sum in (5) contain one term each, for i = 1. The second product runs from 2 to 1; in other words it is empty, and is treated as the identity. Thus (5) reduces to $v_j^1 = g(1) * v_j^0 + h(1)$, which is just (4) with i = 1.
Induction step. Assume (5) holds for m. Then

$$v_j^{m+1} = g(m+1) * v_j^m + h(m+1)$$

$$= g(m+1) * \left(v_j^0 * \prod_{i=1}^{m} g(i) + \sum_{i=1}^{m} \left(\left(\prod_{k=i+1}^{m} g(k) \right) * h(i) \right) \right) + h(m+1)$$

$$= v_j^0 * \prod_{i=1}^{m+1} g(i) + g(m+1) * \left(\sum_{i=1}^{m} \left(\left(\prod_{k=i+1}^{m} g(k) \right) * h(i) \right) \right) + h(m+1)$$

$$= v_j^0 * \prod_{i=1}^{m+1} g(i) + \sum_{i=1}^{m} \left(\left(\prod_{k=i+1}^{m+1} g(k) \right) * h(i) \right) + h(m+1)$$

$$= v_j^0 * \prod_{i=1}^{m+1} g(i) + \sum_{i=1}^{m+1} \left(\left[\prod_{k=i+1}^{m+1} g(k) \right] * h(i) \right)$$

In the next to last step, the g(m+1) distributes over the sum. The last step is valid because when i=m+1, the product has no terms, so $\left[\prod_{k=i+1}^{m+1} g(k) \right] * h(i)$ is just h(m+1). This completes the proof of the theorem.

4.3 Special Cases

The solution (5) is very complex and not likely to be useful for proving anything in its full form. Its advantage is that it is a general, closed-form solution that can be derived automatically; and many cases that do arise in practice can be derived from it. Table

g(i)	h(i)	$\hat{f}_j(m, v_j^0)$
0	k	k
1	k	$v_j^0 + m*k$
k	0	$v_j^0 * k^m$
1	h(i)	$v_j^0 + \sum_{i=1}^{m} h(i)$
g(i)	0	$v_j^0 * \prod_{i=1}^{m} g(i)$

Table 1. Special Cases of the General Solution

1 shows a number of these cases.

`Audit` can handle loops with counters, with simple arithmetic, such as addition and constant multiplication, and with combinations of these operations. It can also handle loops that read or write sequences of values and that combine or store them in various ways. If a loop does not have the proper form for `audit`'s analysis, it is necessary to decompose the verification procedure by writing more detailed CPA specifications for parts of the loop that have the requisite form.

4.4 The Analysis Procedure

To analyze the paths originating at a given node in the state transition graph of an efsm, audit first does a depth-first search from that node to identify all loops on those paths. Then it analyzes the loops, starting with the innermost ones. For each loop it gives the relevant registers symbolic initial values and goes through the loop once, using symbolic simulation to find the transformation applied to each register, along with the conditions that must be satisfied to stay in the loop and the input/output

events executed in the loop. Then, if the register transformations are of the form required in equation (4), they are generalized as in equation (5). The conditions and input/output events are also generalized. For example, if the condition from one iteration is $X[i_0] < k$, where X is an input port and i is incremented by one in the loop, the condition would generalize to: @i $i_0 \le i < i_0 + m$. $X[i] < k$.

When a symbolic simulation reaches the head of a loop that has been analyzed, the generalized register transformations, conditions, and input/output events are added as if they had occurred on a single transition, and the simulation continues along all paths leaving the loop.

5. EXAMPLES

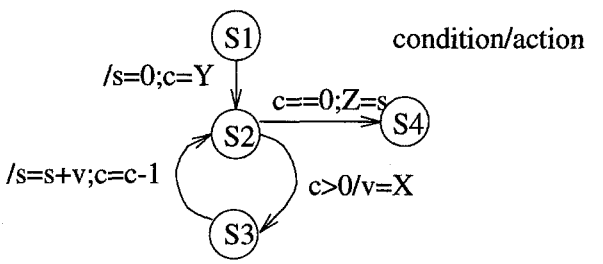

Figure 4. Efsm to Sum Inputs

Figure 4 shows an efsm that sums a set of inputs read at input port X. The number of inputs is read at input port Y, and the sum is written at output port Z. c and s are registers, with c used as a counter and s as the accumulator for the sum. The register v is simply used to hold the value input at X. A CPA event expression that describes the intended behavior is

```
when TRUE {Y --> Z[+1]}
  where Z[+1].v == (apply + with i (i >= 1 && i <= Y[0].v) {X[i].v});
```

where (apply + with i (i >= 1 && i <= Y[0].v) {X[i].v}) means $\sum_{i=1}^{Y[0]} X[i]$. After analyzing the path from S1, where Y is read, to S4, where Z is written, including the loop, audit gives the input/output relation

$$Z[1] == \sum_{i=1}^{Y[1]} X[i]$$

With some index translation, done automatically by audit, this satisfies the specification.

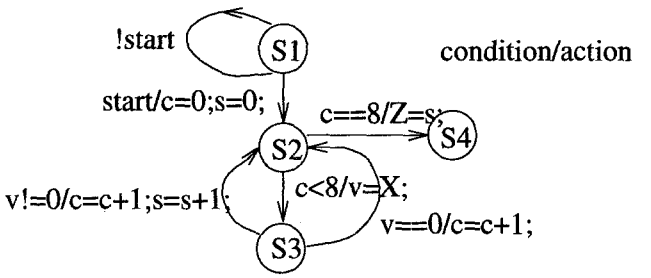

Figure 5. Efsm to Count 1s

The efsm in figure 5 idles until it sees a 1 on the start input, then counts the number of nonzero values in the next 8 inputs at X, putting the count out on the Z output. When asked to analyze the behavior from the occurrence of 1 at the start input to the output at Z, audit finds the input/output condition $Z[1] == \sum_{i=1}^{8}(X[i]==0?0:1)$, where b?x:y is an expression that means "if b then x else y". In other words, the output at Z is the count of cases where X[i]!=0. This conforms to the specified behavior.

Each of these machines took a few milliseconds to analyze. The time was completely independent of the sizes of the registers.

5.1 Current Status and Future Work

At this writing, most of the coding for `audit` has been completed. The one piece missing is the part that translates from `audit`'s internal representations of facts to a form usable by Otter. So far we have been doing that by hand. Once we finish that, we need to do more testing of Otter to see what its limits are for solving the problems given it by `audit` and what facts and guidance it needs to do an efficient job. We also need to make the code for efsm analysis more robust and more general so that it can handle larger and more realistic examples.

When the initial version of `audit` is completed, we need to expand it to handle many of the more complex features of CPA, including parallel events, hierarchical events, and so on. We also need to explore the problem of efficiency, testing `audit` on some substantial examples to probe its limits and the factors to which it is most sensitive. We can certainly improve its efficiency by doing more to exploit the locality of event expressions, by finding parts of the analysis that can be shared, and by finding more efficient representations.

Finally, we want to develop a method for verifying one CPA description against another. This would allow a hierarchical verification methodology, which would be

very important for the verification of large systems.

We have shown that CPA, abstract and rich as it is, can be used as a basis for verification. We have also shown how the symbolic treatment of the data state and symbolic loop analysis can be used to avoid the state explosion problem. This comes at the cost of a more complex form of analysis, which results in decision problems that are semi-decidable. We need to do more experimentation to understand fully the implications of this approach.

REFERENCES

[1] Bochmann, G. V., "Hardware Specification with Temporal Logic: An Example," *IEEE Transactions on Computers* **C-31**(3), pp. 223-231 (March, 1982).

[2] Browne, M. C., Clarke, E. M., Dill, D. L., and Mishra, B., "Automatic Verification of Sequential Circuits Using Temporal Logic," *IEEE Transactions on Computers* **C-35**(12), pp. 1035-1044 (December, 1986).

[3] Darringer, J., "The application of program verification techniques to hardware verification," *Proceedings of the Sixteenth Design Automation Conference*, pp. 375-381, ACM SIGDA and IEEE Computer Society DATC (June, 1979).

[4] Crocker, S., "State deltas: A formalism for representing segments of computation," UCLA-ENG-7784, Computer Science Department, University of California at Los Angeles (February, 1978).

[5] Hunt, W. A., *FM8501: A Verified Microprocessor*, PhD Thesis, University of Texas at Austin (February, 1986).

[6] Devadas, S., Ma, H. T., and Newton, A. R., "On The Verification of Sequential Machines At Different Levels of Abstraction," *Proceedings of the 24th Design Automation Conference*, pp. 271-276, ACM/IEEE (June, 1987).

[7] Kurshan, R. P. and McMillan, K. L., "Analysis of Digital Circuits Through Symbolic Reduction," *IEEE Transactions on Computer-Aided Design* **10**(11), pp. 1356-1371 (November, 1991).

[8] Berthet, C., Coudert, O., and Madre, J., "New Ideas on Symbolic Manipulations of Finite State Machines," *International Conference on Computer Design*, IEEE (September, 1990).

[9] Bryant, R., "Graph-Based Algorithms for Boolean Function Manipulation," *IEEE Transactions on Computers* **C-35**(8), pp. 677-691 (August, 1986).

[10] Burch, J. R., Clarke, E. M., and Long, D. E., "Representing Circuits More Efficiently in Symbolic Model Checking," *Proceedings of the 28th Design Automation Conference*, pp. 403-407, ACM/IEEE (June, 1991).

[11] Gordon, M., "Why higher-order logic is a good formalism for specifying

and verifying hardware," in *Formal Aspects of VLSI Design*, Milne, G. and Subrahmanyam, P. A. (Eds.), North Holland, New York (1986).

[12] Wegbreit, B., "The synthesis of loop predicates," *Communications of the ACM* **17**(2), pp. 102-112 (February, 1974).

[13] Devadas, S., Keutzer, K., and Krishnakumar, A. S., "Design Verification and Reachability Analysis Using Algebraic Manipulation," *Proceedings of the International Conference on Computer Design*, pp. 250-258, IEEE (October, 1991).

[14] Lee, D. and Yannakakis, M., "Online Minimization of Transition Systems," *Proceedings of the 24th Annual ACM Symposium on the Theory of Computing*, pp. 264-274, ACM (May, 1992).

[15] Burch, J. R., Clarke, E. M., McMillan, K. L., Dill, D. L., and Hwang, J., "Symbolic Model Checking 10^20 States and Beyond," *Proceedings of the Fifth Annual Symposium on Logic in Computer Science* (June, 1990).

[16] McFarland, M. C. and Kowalski, T. J., "Specifying System Behavior in CPA," *ICCD91*, pp. 342-345, IEEE (October, 1991).

[17] Chu, T., *Synthesis of Self-timed VLSI Circuits from Graph-Theoretic Specifications*, PhD Thesis, Department of Electrical Engineering and Computer Science, MIT (May, 1987).

[18] McFarland, M. C., "CPA: Giving an Account of Timed System Behavior," *ACM International Workshop on Timing Issues in the Specification and Synthesis of Digital Systems (TAU'90)*, ACM, Vancouver, BC (August, 1990).

[19] McFarland, M. C., Kowalski, T. J., and Peman, M. J., "Language and Formal Semantics of the Specification System CPA," *Proceedings of the International Workshop on Hardware-Software Co-design*, IEEE, Estest Park, CO (September 30, 1992).

[20] King, J. C., "Proving programs to be correct," *IEEE Transactions on Computers* **C-20**(11), pp. 1331-1336 (November, 1971).

[21] Sarkar, D. and De Sarkar, S. C., "Some Inference Rules for Integer Arithmetic for Verification of Flowchart Programs on Integers," *IEEE Transactions on Software Engineering* **15**(1), pp. 1-8 (January, 1989).

[22] Wos, L., Overbeek, R., Lusk, E., and Boyle, J., *Automated Reasoning: Introduction and Applications*, McGraw-Hill, New York (1991).