SPECIFICATION AND AUTOMATIC VERIFICATION OF SELF-TIMED QUEUES*

David L. Dill
Steven M. Nowick
Robert F. Sproull

Technical Report: CSL-TR-89-387

August 1989

This work has been supported under NSF grant # MIP-8858807 and by Apple Computer, Austek Mirosystems, Digital Equipment Corp., Evans and Sutherland, Floating-Point Systems, and Schlumberger as part of the Asynchronous Systems Project.

SPECIFICATION AND AUTOMATIC VERIFICATION OF SELF-TIMED QUEUES*

David L. Dill
Steven M. Nowick
Robert F. Sproull

Technical Report: CSL-TR-88-387

August 1989

Computer Systems Laboratory
Departments of Electrical Engineering and Computer Science
Stanford University
Stanford, CA 94305

Abstract

Speed-independent circuit design is of increasing interest because of global timing problems in VLSI. Unfortunately, speed-independent design is very subtle. We propose the use of state-machine verification tools to ameliorate this problem. This paper illustrates issues in the modelling, specification, and verification of speed-independent circuits through consideration of self-timed queues. User-level specifications are given as Petri nets, which are translated into trace structures for automatic processing. Three different implementations of queues are considered: a chain of queue cells, two parallel chains, and "circular buffer" example using a separate RAM.

Kev Words and Phrases: asynchronous, self-timed, speed-independent, concurrent, hardware verification, finite-state, concurrent system, queue, buffer Petri net.

Specification and Automatic Verification of Self-Timed Queues[*]

David L. Dill Steven M. Nowick
Computer Systems Laboratory
Stanford University
Stanford, CA 94305

Robert F. Sproull
Sutherland, Sproull & Associates
4516 Henry St.
Pittsburgh, PA 15213

1 Introduction

Asychronous (unclocked) designs are of increasing interest because of the cost of broadcasting clocks over large areas of a chip. However, asynchronous circuits are more difficult to design than synchronous circuits. While a synchronous circuit can often be regarded as a single process, with all of the components executing in lock-step, an asynchronous circuit is best thought of as a concurrent system — each component is a separate process that communicates with its neighbors by signalling on wires. Such a design must work in spite of timing variations among the components. One way to expedite the design of correct asynchronous circuits is to use an automatic verifier.

There are two major problems in automatic verification: implementing it efficiently, and formally describing the problem to be verified There is an existing automatic verifier for asynchronous control circuits, which has been described elsewhere [3]. Petri nets are an especially appropriate formalism for describing the behavior of such circuits [14, 10, 11, 9, 1]. In this report, we show how Petri nets can be used as an effective user-level description language in automatic verification.

The techniques described here are highly effective for ***control*** circuits. Other hardware verifiers [5, 8] are not appropriate for such circuits, which are often concurrent and nondeterministic. To illustrate the effectiveness of the verifier, we focus on an important set of examples: self-timed queues.

In the body of the report, we first use Petri nets to describe the handshaking protocol used by a class of self-timed queues. We then describe the theory and implementation of the existing verifier, and a new Petri net front-end. ***The*** verifier is based on ***trace theory,*** and the Petri net descriptions are translated into equivalent trace-theoretic representations.

The system is then applied to a series of progressively more involved queue implementations. Our final example is a non-obvious queue implementation that uses RAM storage. All of the components are described using Petri nets, as is the high-level specification. The verifier detects an error in this implementation; we then present and verify a corrected implementation.

1.1 Self-timed queues

In a sequence of processing modules with varying processing times, throughput can be improved by inserting queues between the modules. It is important for queues to be very fast in order to minimize latency. The speed of a synchronous queue is limited by the speed of the clock, so asynchronous implementations are attractive.

[*]This work was supported by the National Science Foundation under grant number MIP-8858807 and by Apple Computer, Austek Microsystems, Digital Equipment Corp., Evans and Sutherland, Floating-Point Systems, and Schlumberger as part of the Asynchronous Systems Project

A self-timed ***queue*** consists of a data input and data output (consisting of many wires, perhaps). Each of these has associated control wires that indicate when the data is valid and when it is no longer needed. Various protocols can be used for these signals. A particularly simple one is ***two-phase handshaking,*** which uses a pair of wires called ***request*** and ***acknowledge*** on both sides of the queue (on the input side, these wires are labeled *rin* and *ain*, and on the output side they are labeled rout and *aout*).

In two-phase handshaking, it is convenient to ***think*** in terms of ***transitions,*** that is, changes between logical 0 and 1, instead of logical values on the wires (sometimes two-phase signalling is ***called transition signalling***). A transition on a request wire indicates that data is available, and a transition on an acknowledge wire indicates that data has been "seen" and stored (so it is no longer required to be stable). The sequencing of signals is: the data becomes valid (all of the wires carrying the data assume their desired values and remain stable); then there is a transition on the r wire, indicating that data should be loaded into or removed from the queue; finally, the data value remains stable until a transition occurs on the a wire, indicating that the data is no longer needed.

An important property of queues is that they can be concatenated to make longer queues. If a queue having the capacity to store m items (called ***an m-queue)*** is connected to a queue of capacity n, the result is a queue with capacity $m + n$. Hence, a queue of any length can be constructed by concatenating an appropriate number of one-queues. Henceforth, a one-queue is also called a ***stage.***

1.2 Petri net specifications

Even for a simple circuit, such as a queue stage, informal specification can be confusing and imprecise. Informal specifications are especially unsuitable for automatic processing. Instead, ***Petri nets*** can be used for more precise specification. There is a long tradition of using Petri nets for formal specification of speed-independent circuits [14, 10, 11, 9, 1].

We summarize Petri nets only briefly, in part to establish terminology. Detailed discussions of Petri nets are available elsewhere [12]. A Petri net consists of finite sets of ***places, bars***[1] and ***arcs. An arc*** always connects a place p to a bar b or a bar b to a place p; ***in the*** first ***case,*** p ***is*** called ***an input place of*** b, and in the second, p is ***called an output place of*** b. In a diagram of a Petri net, places are depicted as circles, bars as straight lines, and arcs as arrows between the places and bars.

A ***marking*** of a Petri net is an assignment of numbers of ***tokens*** to the places in the net. Intuitively, a marking is a "state" of the net. A marking is depicted by appropriate numbers of black dots in the places of the net.

In a particular marking, a bar is said to ***be enabled*** if all of its input places have tokens. An enabled bar may ***fire,*** removing one token from each input place and adding a token to each output place. If there are k arcs from a place to a bar, there must be at least k tokens on the place for the bar to be enable; when it fires it removes k tokens from the place. Likewise, if there are k arcs from a bar to an output place, firing the bar adds k tokens to the output place.

A ***firing sequence*** is an alternating sequence of markings and bars, beginning with an initial marking. Each marking must enable the bar that follows it in the sequence, and firing that bar must result in the next marking in the sequence. A marking ***is reachable*** from a given marking if it is an element of some firing sequence that begins with the given marking.

Our Petri nets have two additional properties. First, each bar is labeled with a wire name. The firing of a bar represents a signal transition on the wire which labels it. Thus, the sequence of wire names which correspond to the bars in a firing sequence represents a possible history of wire transitions. Second, our Petri nets are ***bounded,*** meaning that only a finite set of markings are reachable from the initial marking. This restriction ensures that the specification is finite-state.

A Petri net specification for a queue stage appears in figure 1. Intuitively, the left and right halves of the net enforce two-phase handshaking behavior for the input and output side of the stage. Additionally, the net ensures that each ***rout*** transition is preceded by a corresponding ***ain*** transition (the cell must have stored its new input data before ***rout*** occurs, signalling that new data is available on the output). It also requires that no two ***ain*** transitions

[1]Usually called *transitions*, but we have already used that term for changes in wire values.

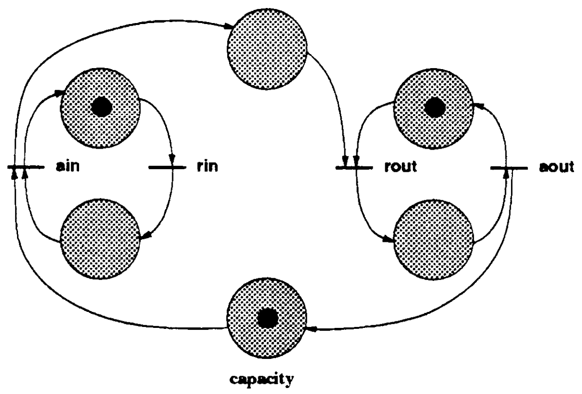

Figure 1: Petri net for queue stage.

can occur without ***an aout*** transition occuring between them (the queue cannot accept new data until its old data has been removed).

Note that this specification does not give the behavior of the data values. Our verification method is not generally appropriate for handling data values — it ***is*** better for ***control circuits.*** We believe that the problems of verifying data computations and control signals are fundamentally different, and that it is best to apply different methods to each. Hence, this paper will concentrate on the verification of control signals, which is in fact the subtlest problem in our examples.

One of the advantages of Petri net specifications is ***scalability.*** For example, if the Petri net for a queue stage is properly chosen, it is easy to modify it to specify a queue of capacity k, by putting k tokens in the place labeled ***capacity*** in figure 1. The added tokens allow ***ain*** to fire k times before ***aout*** has fired, modeling the fact that k items may be stored in the queue before ***any*** are removed from the other side ***(as*** signalled by *aout*). ***In*** some other work, Petri nets have been required to be ***safe;*** that is, in each reachable marking of a Petri net, there may be ***at most*** one token per place[1, 9]. As the above example shows, this requirement of safeness negates an important advantage of Petri nets as a description language.

Any model of concurrency should answer certain questions: When are two behaviors equivalent? How do concurrent behaviors interact? When does one behavior implement another? The next section provides the answers by ***giving*** Petri nets a ***trace theory*** semantics. More importantly, it provides a way of automatically verifying with Petri nets.

2 Trace Theory on Finite Automata

Trace theory is a formalism for modeling, specifying, and verifying speed-independent circuits. Speed-independent circuits are those which function correctly assuming arbitrary delays in the components and no delays in the wires.

Trace theory is based on the idea that the behavior of a circuit can be completely described by a regular set of ***traces,*** or sequences of transitions. A single trace corresponds to a partial history of signals that might be observed at the input and output wires of the circuit.

The idea of using trace models of concurrent systems appears to originate with Hoare, who gave a trace semantics for a subset of CSP [6]; the specific application to asynchronous circuits was first proposed by Rem, Snepscheut, and Udding [13]. The refinement used here is by Dill [2, 3].

This section provides a brief summary of trace theory based on finite automata (we assume familiarity with regular languages and finite automata [7]). Alternative presentations of trace theory are available in more detail elsewhere [2, 3].

2.1 Trace structures

A ***trace structure*** is a triple $\mathcal{T}$ ***= (I.** $O.\mathcal{M}$***),*** where ***I*** is a finite set of ***input wire names,*** O is a ***finite*** set of ***output wires,*** and $\mathcal{M}$ is a deterministic finite automaton (DFA). I and O must be disjoint. We use the abbreviation A for $I \cup O$. The automaton is a quadruple ***(* Q. n.** $q_0 . Q_F$ ***),*** where **Q** is a finite set of ***states,*** n : **Q** x A — **Q** (which can be a partial function) ***is the next-state function,*** $q_0 \in$ **Q** is the ***start state,*** and $Q_F \subseteq Q$ is the set of ***final*** states. In a trace structure, Q_F is always the same as **Q;** however, some of the automata produced by intermediate steps of the constructions below have **Q** $_F \neq$ **Q.**

The regular language accepted by $\mathcal{M}$ ***is the*** set of ***successful traces,*** written S, which are the partial histories the circuit can exhibit when "properly used"[2]. If it is improperly used, the results are not specified. In this way, a trace structure is like a warranty that can be voided if certain conditions arc not met. A conventional example of a proper-use constraint is the requirement that the ***set*** and ***reset*** inputs of a latch never be asserted simultaneously.

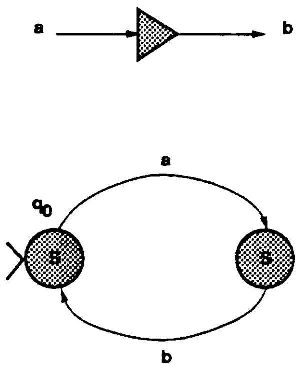

Figure 2: Non-inverting Buffer and Trace Automaton.

Consider a trace description of a non-inverting buffer (figure 2). The trace description depends on the initial conditions of the circuit; in this example, the logical values on the input and output wires are initially the same. The buffer waits until an input transition a occurs, then produces an output b, and so on. The undefined "next states" in the state diagram are as interesting the defined ones: in the initial state there is no successor for b because it is ***impossible*** for a b transition to occur — b is an output, so it is under the control of the circuit. The missing successor on a in the next state models a ***possible but undesirable*** occurrence. It is possible because the input to a gate may change at any time (the gate cannot prevent it from happening). It is undesirable because two consecutive input changes may cause unpredictable output behavior — the output may pulse to the opposite value for an arbitrarily short time, or it may not change (because the input pulse was too short), or it may change "halfway" and then return to its original value. Such behavior in a speed-independent circuit is unacceptable, so our description of a buffer forbids inputs that can cause it.

The possible but undesirable ***traces are*** called ***failures.*** The set of all failures associated with a trace structure $\mathcal{T}$ is written F. An automaton accepting F can be derived from $\mathcal{M}$ by adding a failure state q_F, making q_F the sole accepting state (so $Q_F = \{ q_F \}$), and adding successors according to the following rules:

- For ***every input wire a*** and state q, if **n**(q, a) is undefined, add a transition from y on a to y $_F$.
- Add a transition from q_F to itself on every input and output.

The first rule means that if there is no transition on an input symbol from a state, the circuit is not "ready" for that input — a failure will result if it occurs. The second rule says that a failure cannot be redeemed by the arrival of additional inputs and that the output behavior is completely unpredictable — any output may occur at any time.

[2]**Note that the case where $S = \emptyset$ cannot be represented in the automaton as defined above, for trivial reasons. This can easily be dealt with as a special case, so we will not discuss it further.**

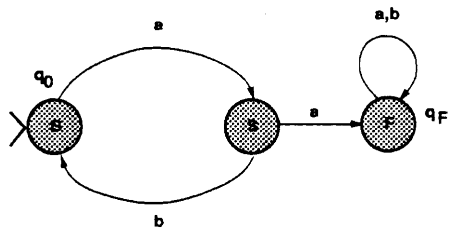

Figure 3: Trace Automaton with Failure State for Non-inverting Buffer.

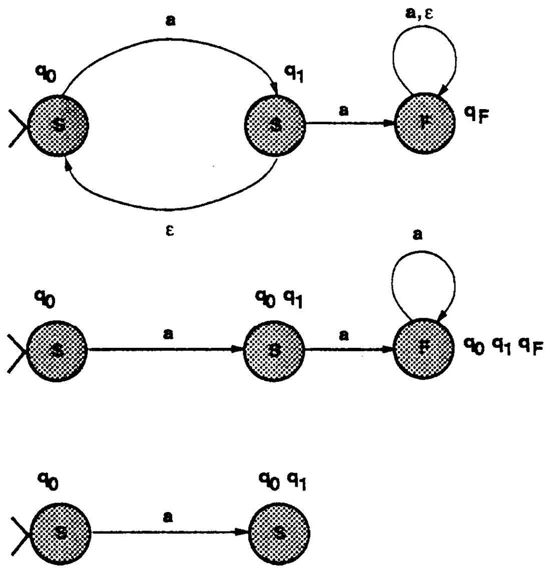

Figure 4: Eliding the Output of a Non-inverting Buffer.

The automaton in figure 3 accepts the failure set of a buffer, where the final state is q_F (any String reaching a state marked S is a success; any reaching F is a failure). The failures are the traces that include the substring a a.

2.2 Operations on trace structures

There are three fundamental operations on trace structures: **hide** makes some output wires unobservable, **compose finds** the concurrent behavior of two circuits that have had some **wires** connected, and **rename** changes the names of the wires.

hide takes two arguments: a set of wires to delete and a trace structure. The deleted wires must be outputs, otherwise the operation is undefined An automaton for the result of **hide** can be obtained by **first** adding a failure state (as above), then mapping each of the hidden wire names to ϵ, the empty string, to give a nondeterministic finite automaton with ϵ transitions. This automaton can be converted to a deterministic automaton by using the subset construction with ~-closure (see [7]). Every state in the resulting automaton whose associated subset contains q_F should then be deleted, giving the desired automaton.

Figure 4 shows, step-by-step, the effects of hiding b in the buffer example of figure 3. This example **illustrates** the justification for the last step. Consider the trace aa when b is hidden. This could reflect several possible traces in the original circuit. It could be the a transitions of the trace aba — a success. Or it could be the a transitions of aa — a failure in the original circuit. If the inputs aa are sent to the circuit without regard to b, the choice between success or failure is nondetenninistic. For the purpose of implementing a specification, a circuit that "might fail" when given two a inputs is as bad as a circuit that "always fails" given those inputs. In the construction of the previous paragraph, the states representing "possible failure" are those whose subsets contain q_F. Deleting these states has the desired effect of classifying as failures the traces that reach them.

In this example, the successes are exactly ϵ and a. Intuitively, there is no safe way to send more than one a to a buffer unless the output can be observed, to ensure that a b occurs between each pair of a's. (One can reasonably conclude that a non-inverting buffer with hidden output is quite useless.)

hide is an important operation because it automatically suppresses irrelevant details of the circuit's operation — the unobservable signals on internal wires. In practice, hiding internal signals can result in a smaller model of behavior. It is also important because it allows trace structures to be compared based on their interface signals while ignoring internal signals. The ability to hide is one of the advantages of trace theory that is missing from earlier efforts [4].

compose models the effect of connecting identically-named wires between two ***circuits*** (called ***the components).*** Two circuits can be composed whenever they have no output wires in common. The composition of $\mathcal{T}$ and $\mathcal{T}'$ can be written $\mathcal{T} \mid \mathcal{T}'$. If $\mathcal{T}'' = \mathcal{T} \mid \mathcal{T}'$, the set of outputs of $\mathcal{T}''$ is $O'' = O \cup O'$ (whenever an output is connected to an input, the result is an output) and the set of inputs is $I'' = (I \cup I') - O''$ (an input connected to an input is an input). Note that $A'' = A \cup A'$.

The construction of $\mathcal{M}''$, the automaton of the composite trace structure, consists of two steps. The first step is to define an automaton $\widehat{\mathcal{M}}$ using a modified product construction. First, add the failure state q_F to $\mathcal{M}$ and q'_F to $\mathcal{M}'$, as in the definition of **hide.** The states in $\widehat{\mathcal{M}}$ are pairs of states in $\mathcal{M}$ and $\mathcal{M}'$: $\hat{Q} = (Q \times Q')$ and the start state, $\hat{q}_0$, is (q_0, q'_0). The definition of the successor function involves several cases:

$$\hat{\mathbf{n}}(q_i, q'_j) = \begin{cases} (\mathbf{n}(q_i, a), q'_j) & \text{if } a \in A - A' \\ (q_i, \mathbf{n}'(q'_j, a)) & \text{if } a \in A' - A \\ (\mathbf{n}(q_i, a), \mathbf{n}'(q'_j, a)) & \text{otherwise} \end{cases}$$

In effect, each component ignores wires that are not its inputs or outputs. If either of $\mathbf{n}$ or $\mathbf{n}'$ is used in the definition of the relevant case and is undefined, $\hat{\mathbf{n}}(q_i, q'_j)$ is undefined, also.

The second Step is to delete every State that can reach a State of the form (q_i, q'_F) or (q_F, q'_j) by a string of zero or more outputs (the rationale for this is explained in the example below). This gives the final automaton.

Figure 5 shows the initial step in the composition of two noninverting buffers in series, and figure 6 shows the final result. The automaton for the first buffer is as in figure 3; the automaton for the second buffer is the same, except that b and c are substituted for a and b. To justify the last step in the construction, consider the state (q_1, q'_1), which is a success as shown in the figure. The state is a success, but whenever it is entered there is a possibility that the first component will output a b, causing a failure in the second component. For the user of this circuit to guarantee that no failures can occur, (q_1, q'_1) must be avoided — in effect, it should be classified directly as a failure. Intuitively, an internal failure can be exported to the interface of the circuit.

rename takes as arguments a trace structure and a renaming function which maps its wire names to new names. The function must be a one-to-one correspondence — it is illegal to rename two different wires to be the same. The effect of **rename** on an automaton is simply to substitute the new names for the old names. This operation is useful for creating an instance of a circuit from a prototype.

2.3 Verification with trace theory

A trace structure specification of a circuit can be compared with a trace structure description of the actual behavior of the circuit, just as a logical specification can be compared with a program implementing it. When $\mathcal{T}_I$ implements $\mathcal{T}_S$, we say that $\mathcal{T}_I$ ***conforms to*** $\mathcal{T}_S$, that is, $\mathcal{T}_I \preceq \mathcal{T}_S$ (the inputs and outputs of the two trace structures must also be the same). This relation holds when $\mathcal{T}_I$ ***can be safety substituted*** for $\mathcal{T}_S$. More precisely, $\mathcal{T}_I \preceq \mathcal{T}_S$ if, for every $\mathcal{T}'$, whenever $\mathcal{T}_S \mid \mathcal{T}'$ has no failures, $\mathcal{T}_I \mid \mathcal{T}'$ has no failures, either. Intuitively, $\mathcal{T}_I$ must be able to handle every input that $\mathcal{T}_S$ can handle (othexwise, $\mathcal{T}_I$ could fail in a context in which $\mathcal{T}_S$ would have succeeded), and must not produce an output unless $\mathcal{T}_S$ produces it (otherwise, $\mathcal{T}_I$ could cause a failure in the surrounding circuitry when $\mathcal{T}_S$ would not).

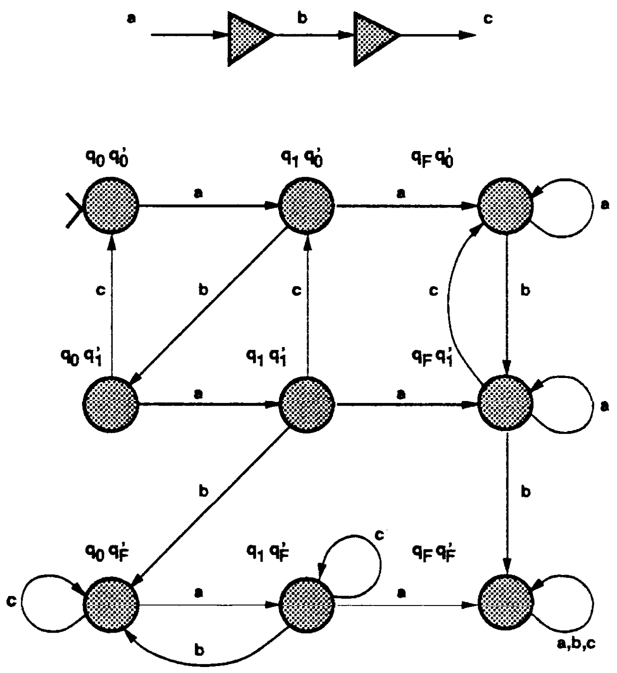

Figure 5: Serial Composition of Non-inverting Buffers. First step.

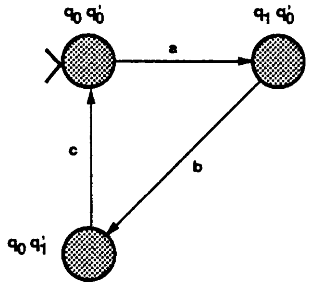

Figure 6: Serial Composition of Non-inverting Buffers. Final result.

This relation can be tested by exchanging the input and output ***sets*** of $\mathcal{T}_S$ to obtain $\mathcal{T}_S^M$ ***(the mirror*** of $\mathcal{T}_S$), in which $O_S^M = I_S$ and $I_S^M = 0_S$, then composing $\mathcal{T}_I \mid \mathcal{T}_S^M$ and checking whether the failure set of the composite trace structure is empty. This result is proved and justified in detail elsewhere [2]. The intuition behind this result is that $\mathcal{T}_S^M$ represents a context that **will** "break" any trace structure that is not a true implementation of $\mathcal{T}_S$. Specifically, $\mathcal{T}_S^M$ produces as an output everything that $\mathcal{T}_S$ accepts as an input, so if $\mathcal{T}_I$ fails on any of these, there will be a failure in $\mathcal{T}_I \mid \mathcal{T}_S^M$. Similarly, $\mathcal{T}_S^M$ accepts as input only what $\mathcal{T}_S$ produces as output, so if $\mathcal{T}_I$ produces anything else, there **will** be a failure in $\mathcal{T}_I \mid \mathcal{T}_S^M$, also.

Conformation ***is*** a ***partial order*** rather ***than an equivalence,*** since an implementation ***is*** usually a *refinement* of a specification — there may be **details** of an implementation's operation that are not required by the specification. For this reason, conformation implies that an implementation meets ***or exceeds*** a specification.

There is, however, a concept of equivalence that arises from conformation. Two mutually conforming trace structures ***(7 $\preceq$ 7'*** and ***7' $\preceq$ 7) are*** said to ***be conformation equivalent*** (written 7 $\sim$ 7 '). Intuitively, conformation equivalence means that the implementation meets the specification ***exactly,*** that is, that the two trace structures are **interchangable** in correct designs.

Trace theory has the great advantage of supporting ***hierarchical verification.*** Using trace theory, a hierarchical design can be verified in the same way it is organized: a trace structure specification that has been verified at one level can be used in the description of the implementation of the next higher level. Hierarchical verification can be immensely more efficient than the alternative of "flattening" a hierarchical design. Suppose that a module M consisting of several components is used in a larger design. One way to simplify verification is to write a specification S that expresses only the relevant properties of M, verifying that the implementation of M meets the specification S, then verifying the larger circuit using S instead of the implementation of M. This suppresses irrelevant implementation details that otherwise would make verification of the larger circuit more difficult. Also, M may be repeated many times in the larger design, but it only needs to be verified once using the hierarchical method.

2.4 An automatic verifier

There is an implementation of trace theory in Common Lisp that **allows** individual trace structures to be **defined** in **a** variety of ways. **hide, compose,** and **rename** have been implemented as Lisp functions, as has the conformation relation. This program was used for the remaining examples of the paper.

The program can be used as an automatic verifier. There is a function that checks conformation when given a description of the implementation (consisting of trace structures for the primitives and an expression describing the topology of **the** circuit **using hide, compose,** and **rename)** and a specification (a trace structure for the desired behavior). This function composes the mirror of the specification with the implementation and searches for **failures** in the resulting state graph. If the implementation does not conform to the specification the verifier prints a failure trace to help diagnose the problem. For space and time efficiency, the state graph construction is search-driven: states are generated only when needed This simple implementation trick often saves a tremendous amount of computation when the implementation does not conform, because the program stops when it discovers the ***first*** failure. This is particularly important since buggy circuits usually generate many states representing random activity after the first problem occurs.

The primary burden on the user of this program is defining trace structures for primitives and specifications. To aid the user, various pre-processors are used to translate more concise notations into **DFAs** (the structure of the program is such that it is easy to add pi-e-processors). One of these translates bounded Petri nets (as described above) into trace structures. The states of the DFA are markings of the Petri net, and the start state of the DFA is the initial marking. If q_i is a marking (a state in the DFA) and the bar labeled a is enabled, the successor **n(** q_i. **u)** is the marking that results when the bar fires. This results in a DFA when the Petri net is bounded (so that the number of states is finite) and when there is never more than one bar with the same label enabled at the same time (so that there is a unique **next-state**)[3].

[3]**The second restriction could be eliminated by first translating the Petri net into a nondeterministic finite automaton, in the obvious way, and then transforming it into a deterministic finite automaton using the subset construction. However, we have not needed this generality, and the**

3 Queues and Conformation

This section describes experiments with the trace theory verifier on different implementations and specifications of queues.

3.1 Queues of different sizes

The first experiments are on trace structures derived directly from Petri nets that we have already seen.

As a simple example, the program reports that a two-queue fails to conform to a one-queue, and prints the trace ***rin ain rin ain as*** a diagnostic. The trace ***rin ain rin*** can occur in either; however, it is impossible for the one-queue to respond with ***ain as the*** next event — at this point, the queue capacity of one is used up, so it cannot accept the second input. However, the two-queue ***can*** respond with a second ***ain*** — this is the essence of two-queue-ness.

In terms safe substitution, it is quite possible that a one-queue could be used in a context that would not be ready for the additional ***ain*** — such a context depends on the capacity of the queue being no more than one for correct operation. Substitution of a two-queue will cause havoc.

However, a one-queue ***does*** conform to a two-queue. After any trace, it accepts the same input signals and produces a subset of the outputs. This result may seem counterintuitive. In fact, it demonstrates an important limitation on the expressive power of ***trace theory: it cannot expression Ziveness properties,*** that is, conditions that ***occur eventually*** or ***inevitably.*** Shortening the queues in a speed-independent circuit will never cause unwanted signals to occur, but it ***can*** result in the non-occurrence of desirable signals. In particular, it is well known that this can introduce *deadlock*. For example, imagine a circuit that writes two values before reading two more values, and then repeats. If its inputs are connected to its outputs with a two-queue, it will cycle endlessly. However, if the two-queue is replaced by a one-queue, it will deadlock after inserting the first value in the queue. This cannot be detected by the automatic verification method discussed here[4].

3.2 Series Queue

Although conformation can be checked between arbitrary trace structures, the usual case is that an implementation, described as a composition of primitive trace structures, is compared with a single, monolithic specification. Our first experiment, taking this approach, is to show that two queue stages can be concatenated to form a queue of length two. In this case, a description of the implementation will consist of a trace structure for a single stage (from a Petri net), a description of how to concatenate two stages using **hide, compose,** and **rename,** and a trace structure specifying the behavior of a queue of length two (again, from a Petri net). The verifier reports that, as expected, the implementation conforms to the specification.

Are the specification and implementation conformation-equivalent; that is, does the implementation meet the specification exactly? It is easy to determine this automatically, by checking whether the specification conforms to the implementation (note that this is the reverse of the usual order). Testing this requires that we first ***evaluate the*** implementation expression to obtain a single trace structure modelling it. The verifier reports that the specification conforms to the implementation — the concatenated stages implement the specification for the two-queue exactly.

3.3 Parallel queue

Another way to build a large queue out of smaller queues is to connect them ***in parallel, as*** in figure 7 (the wire names inside the gray components are the original names; the names next to the wires are the result of the renaming operations used in the description of the composite implementation). A ***distribute*** module at the input side distributes the inputs evenly between the parallel queues and a ***collate*** module at the output combines the outputs of the two queues into a single output stream. Petri nets for the distribute and collate modules are shown in figures 8 and 9.

implementation would be slower.

[4]Trace theory can be extended to handle liveness properties in very general ways [2]. However, the results are more complicated and much more difficult to implement.

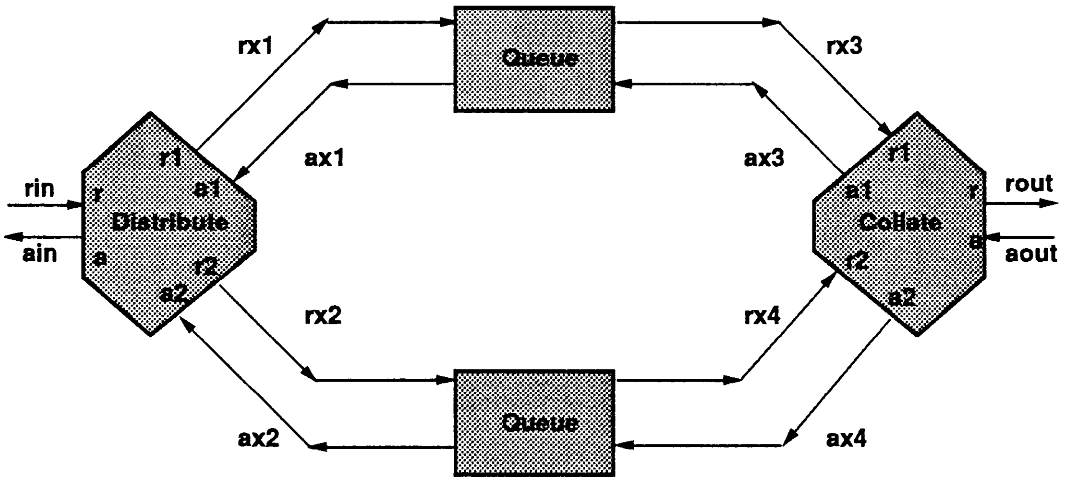

Figure 7: Parallel queue.

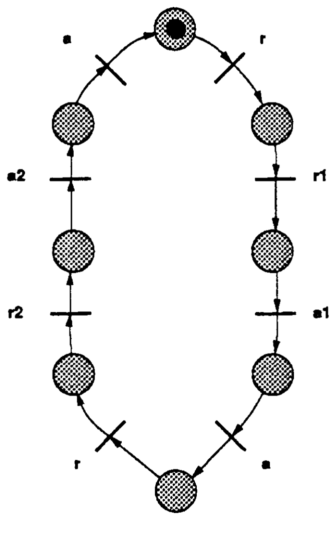

Figure 8: Petri net for distribute module.

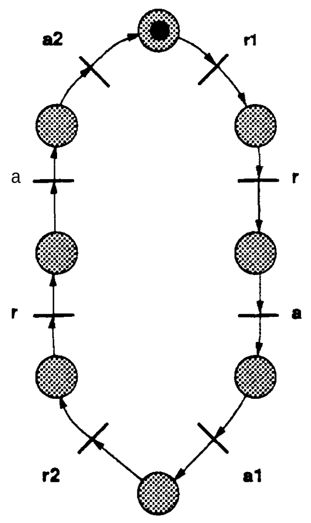

Figure 9: Petri net for collate module.

The experiment in this case is to verify that two one-queues in parallel implement a two-queue. This implementation fails utterly, for the following reason: the concurrency added by the internal queues allows a request on *rx4* to arrive at the collate module before it has sent back ax3 — however, the collate module, as specified, is not ready to receive *rx4* at this point. The problem is that the collate specification restricts the module's inputs unnecessarily. Instead, we want to specify a module that can accept a request on one interface before it has necessarily acknowledged a request on the other. An improved Petri net specification for a collate module appears in figure 10.

When the verifier is applied to the parallel two-queue with the improved collate module, it still fails, printing ***the*** diagnostic trace: ***rin rxl axl rx3 rout.*** The distribute element first receives a request ***rin,*** which it gives to the first queue stage by signalling *rxl*; then the queue stage responds with outputs *axl* and *rx3*. The collate element receives the *rx3* and responds by signalling on ***rout.*** Unfortunately, the specification for a two-queue requires that *ain* precede ***rout in response*** to the first ***rin,*** and no *ain* has occurred yet.

It is quite difficult to find a parallel implementation of the queue that conforms to the given specification. There is, however, a better solution: change the specification! The requirement that *ain* precede ***rout*** is unnecessary. The two signals should be allowed to occur in either order. A Petri net for this specification is shown in figure 11. As with the previous specification, it can be made to specify an n-queue by adding more tokens in the initial marking to the place labeled ***capacity.*** Such specifications are henceforth ***called general queue stages*** and ***general queues,*** respectively.

This change in the specification fixes the problem. The parallel queue using general queue stages conforms to the general specification of a two-queue.

This experiment has demonstrated a well-known principle: ***specifications should be as general as possible.*** Otherwise, the specification may unnecessarily limit the range of implementation alternatives, as in this case. Also, verification and modelling of unnecessary properties can be expensive.

Another point illustrated by this examples is that when an attempt to verify fails, it is often because of a problem in the specification, not the implementation. Verification can be used to debug and improve specifications as well as implementations.

One way to make sure that a specification is not too constrained is to try various implementations, as we have just done. There is at least one other test that is sometimes useful: a specification that is insensitive to wire delays is less likely to fall apart when there are unexpected delays in the implementation — wire delays, or in our example, delays in other components. A trace structure that does not change when wire delays are added is said to be ***delay-insensitive [*** 16, 17]. In trace theory, a wire delay has the same trace structure as a non-inverting buffer, A simple function, called **DI,** can be defined that attaches delays to all of the inputs and output using the

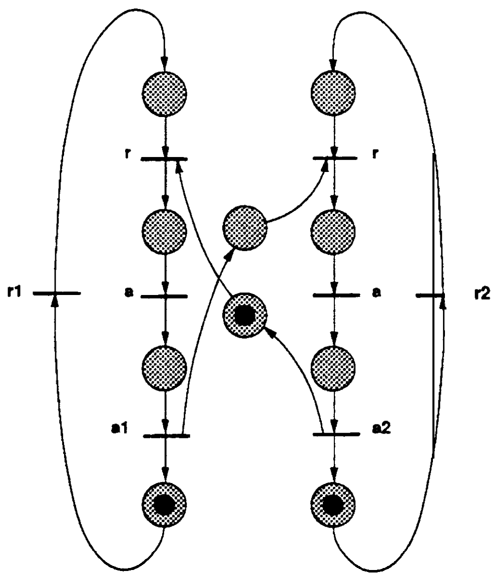

Figure 10: Petri net for improved collate module.

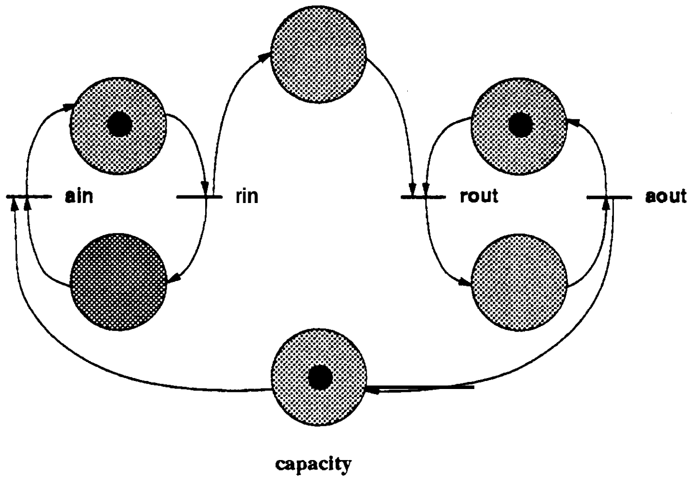

Figure 11: More general queue specification.

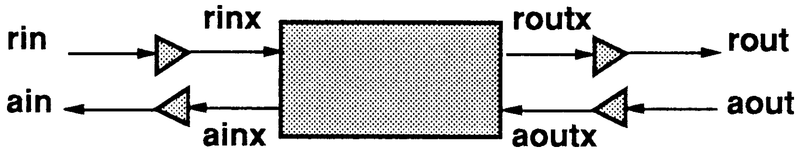

Figure 12: Delay-insensitizing a queue stage.

compose operation and, by appropriate renaming, arranges that the new inputs and outputs have the same names as the original ones, which are hidden (see figure 12). $\mathcal{T}$ is delay-insensitive iff **DI(** $\mathcal{T}$ **)** $\preceq$ **7. (7** $\preceq$ **DI**(**7)** is always true). The program can check this condition. The new stage specification is delay-insensitive, but the old one is not.

Universal use of delay-insensitive trace structures would have the advantage that any implementation that has been verified using them will be robust (continue to meet its specification) in the presence of unexpected wire delays. However, delay insensitivity is a strict property; sometimes it is impossible to achieve, or can only be achieved at the cost of reduced efficiency.

3.4 Hierarchical verification of queues

Up to this point, we have only **verified** that one-queues can be composed in various ways to implement two-queues. Can we verify larger queues? This subsection examines various approaches to this problem. We should point out that, unlike more complex circuits, it is not **difficult** to prove by hand that a chain of n queues implements an n-queue, for all ***n.*** Nevertheless, this is a good example for exploring the limits of the verifier, and for illustrating the effectiveness of hierarchical verification.

In all of the approaches, we need a specification of an n-queue, for various values of n . As was noted earlier, this can be obtained by adding tokens to the Petri net for the one-queue. One limitation on the size of the problems that can be solved is the size of the DFA that is compiled from this Petri net,

Fortunately, the size of the specification grows only linearly in the size of the queue (the exact number of states is 417 + 4 for an n-queue). Intuitively, a state is distingushed by the states of the handshaking interfaces and the number of items in the queue (n possibilities). In practice, we have been able to translate a Petri net for an 1 1 00-queue.

It is not straightforward to derive the number of states examined during verification, so the remaining results are extrapolated from the actual behavior of the program on examples of varying size.

We first verify that a concatenation of n one-queues conforms to a specification of an n**-queue**. The observed behavior of the program is that the number of states grows exponentially. (More precisely, it is defined by the recursive formula $F(n) = 4 \cdot F(\mathbf{n}-1) - F(n-\mathbf{2})$, where $F(\mathbf{n})$ is the number of states. The exact closed-form solution is complicated, but it approximates $(2+\sqrt{3})^{n+1}/\sqrt{3}$ for large n .) Intuitively, the state graph must model the states of the ***n*** – 2 interfaces ***between*** the stages (which are not observable). Since these are loosely coupled, almost all combinations of interface states are represented. The program cannot verify more than n = 6, where it examines **5822 states.**

A better approach would be to take advantage of hierarchical verification: verify that n/2 one-queues conform to an n/2-queue, then verify that two $n/\mathbf{2}$ queues conform to an n-queue. The first task can be done hierarchically, also. Intuitively, the composition of the implementation is almost a product of the state graphs for the two $n/\mathbf{2}$-queues. This strategy makes the growth in states quadratic in n instead of exponential (more precisely, the number of states is $\mathbf{8}(n/\mathbf{2}+1)^2-2$). Using this approach, the program can verify a 64-queue (8710 states).

A better hierarchical approach would be to divide the problem into an (n – 1)-queue and a one-queue. In this ***case, the states grow linearly:*** 1617 – 2. Intuitively, the size of the DFA for the (n – 1)-queue is linear and the size of a one-queue is constant, so the composition is linear, also. **A** queue of size 400 requires 6398 states.

In summary, a verification problem that grows exponentially in the size of the implementation can be cut down

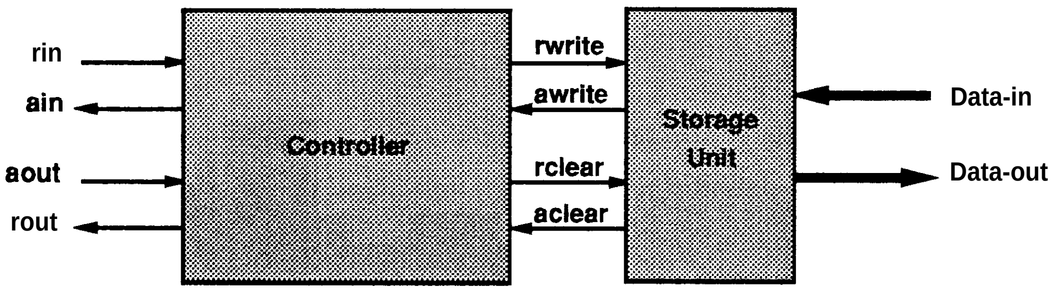

Figure 13: Block diagram of circular buffer queue implementation.

to a linear problem by hierarchical verification. To do this required a formalism (trace theory) that allowed a variety of different problem decompositions and some judgment on our part as to the most efficient decomposition.

4 Circular buffer implementation

The example of this section is a greater challenge to verification because of the variety of components it contains. Unlike many of the previous examples, we believe that this would be very difficult to verify "on paper", and probably quite difficult for symbolic verifiers as well. The implementation is similar to "circular buffers" that are often used in software. Our circuit is based on a design by Sutherland ([15]).

4.1 Implementation description

The implementation of the circular buffer can be divided into two major modules: a storage module and a controller (see figure 13). The storage module handles the actual storage of data, while the controller translates between the queue protocol and the protocol for the storage module.

4.1.1 Storage Module

Internally, the storage module has counters that act as pointers to the head and tail of the queue, random access memory, perhaps read and write buffers, and probably timing-dependent logic to control the signals to the RAM. However, we do not consider the implementation of this module, but instead regard it as a black box and focus on the interface it presents.

The storage module has a data input and data output, which are the data input and data output for the entire queue. The control interface between the modules has two pairs of request/acknowledge wires: ***r-write*** signals that input data should be stored internally and ***awrite*** signals that it has been successfully stored. ***rclear*** indicates that output data is no longer needed and ***aclear*** signals that the next data output is available (it must continue to be available until the next ***rclear).***

There are two special cases to consider. If the storage unit ***is*** empty, ***an awrite*** (following ***an f-write)*** indicates not only that new input data has been stored, but also that the data is now available on the output. Also, in the case where the storage unit holds only one data element, ***an aclear*** (following ***an rclear)*** indicates that the data element has been removed from storage; however, it does ***not*** indicate that new data output is available. There are no constraints on the validity of output data in this case.

In addition, there are some restrictions on the use of the storage unit: first, read and write transactions must be mutually exclusive; second, it is not permissible to signal ***r-write*** unless storage is actually available in the unit; third, it ***is illegal*** to signal ***rclear*** unless the unit actually contains data.

A Petri net for the storage unit control signals is shown in figure 14. It describes the two-phase handshaking protocols for ***r-write*** and ***awrite***, and for ***rclear*** and ***aclear***, using the by now familiar pairs of places and bars.

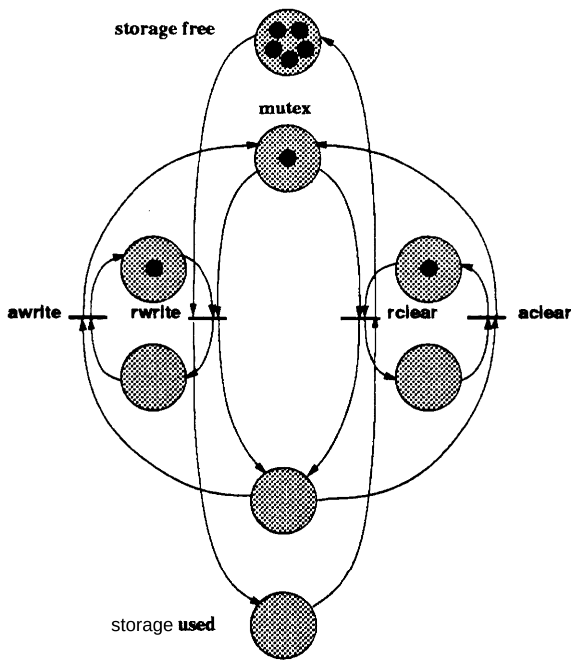

Figure 14: Petri net for storage unit.

Mutual exclusion is enforced by the single token in the place ***mutex.*** If either ***r-write*** or ***rclear*** fires, the token disappears from place ***mutex***, disabling the other request. When the corresponding acknowledge fires, the token is returned to ***mutex***. Note that this is a constraint on the environment: the sequence ***r-write rclear*** is a ***failure.***

The capacity of the storage unit is represented by the number of tokens initially in the place ***storage free.*** Every time ***rwrite*** fires, one of these tokens is removed and added to the place ***storage used; rclear*** returns a token to ***storage*** free. When the storage unit is full ***rwrite*** is disabled, so overfilling will be a failure, also. The place ***storage used*** enforces a similar constraint on clearing.

4.1.2 Controller Module

The other large module in the circular buffer implementation is the controller module, whose purpose is to convert between the queue and storage unit protocols and to protect the storage module from signals that violate its input constraints. Unlike the storage module, we discuss the implementation of the controller in detail.

The controller implementation consists of an arbiter to insure mutual exclusion between write and clear requests to the storage module, an up/down counter to keep track of the number of items in the queue, combinational circuits to test whether the queue is full or empty, and sundry control circuitry (see figure 15). Understanding the detailed operation of this unit may require extended pondering.

The controller contains an internal data path, ***count,*** which carries the current value of the up/down counter. Ideally, it would be possible to describe the implementation at the same level as in figure 15. However, there is no simple way to represent the value of the count as a signal in trace theory. Instead, we model the combination of the up/down counter and the full and empty testers as a single component which we call ***the counter unit. Since*** the ***count*** value is then hidden inside the counter unit, the interface consists entirely of control wires, so the counter unit can be exactly described by a trace structure.

Petri nets tum out to be very convenient for describing the behavior of the counter unit, since we can construct the Petri net for the unit out of “parts” that correspond to the components of the counter unit. There are two places that are shared by each such part of the net, labelled ***count*** and ***n-count.*** The internal value of the count in the

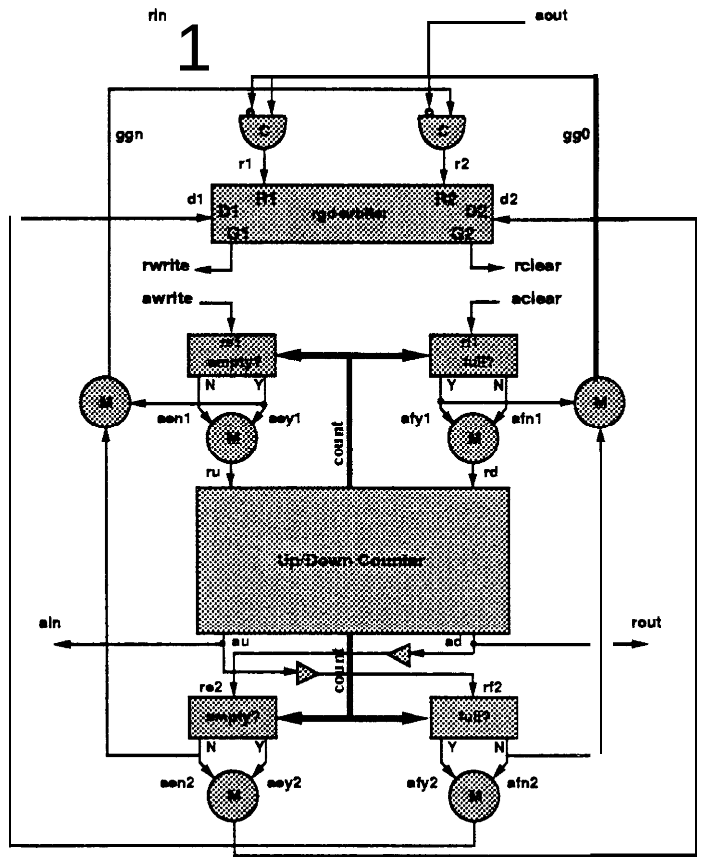

Figure 15: Controller unit.

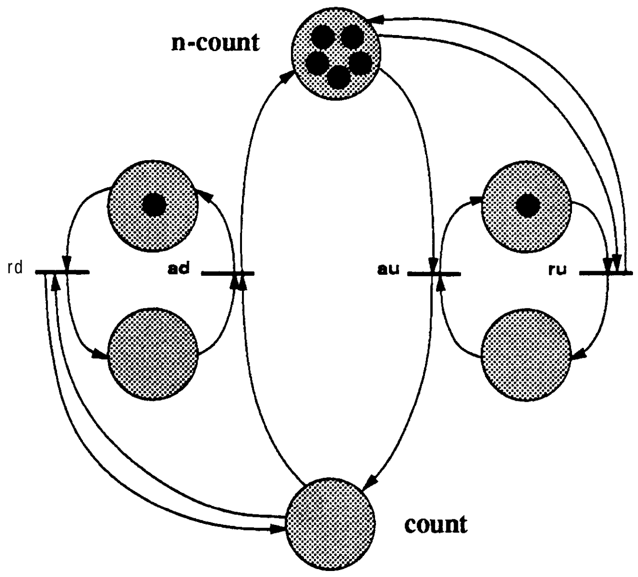

Figure 16: Petri net for up/down counter.

counter unit can be modelled by the distribution of tokens between these two places. This simple decomposition of the net makes it understandable (and drawable!) even though it is quite complicated when taken as a whole.

The first part is a description of an up/down counter, which has four control wires: ***ru,*** which requests an increment in the count; ***au,*** which acknowledges that the count has been incremented; and ***rd*** and ***ad,*** which have the same function for decrementing. The current count (initially 0) is maintained in the place ***count;*** remaining tokens for counting are stored in the place ***n-count.*** These two places are ***similar*** to ***storage free*** and ***storage used*** in the Petri net for the storage unit. The Petri net for the up/down n-counter is shown in figure 16.

Note that this is ***not*** a modulo-n counter. It counts up to n (or down to 0) and then ***fails*** if it receives another ***ru*** (***rd***) request. Our intention is that the counter would never be asked to count above n or below 0; with this description, any violations of this assumption will show up as failures during verification and will be reported.

The Petri nets for the full- and empty- testers share places ***count*** and ***n-count*** with the counter (figure 17). There is a second pair of full/empty testers in the counter unit, also. As a notational convention in these Petri nets, we have attached weights to some of the arcs (numbers next to the arcs). An arc with weight ***n in the*** figure is shorthand for ***n*** corresponding arcs; a bar with such an arc coming into it may not fire until the place at the other end of the arc has at least ***n*** tokens.

Finally, the intended use of the component is that only one type of transaction will be active at a time; none of the requests is allowed to occur if another request is outstanding (has not been acknowledged). As in the storage unit, this condition can be enforced by adding a place with a single token that is an input to all of the request bars and an output of all of the acknowledge bars. The Petri net for the entire unit consists of the Petri nets for the up/down counter, two empty-testers, and two full-testers, with the added place and arcs to enforce mutual exclusion as described above.

The remaining components in the controller are asynchronous building blocks that one might expect to use in other designs. These include an arbiter, merge elements, C elements and non-inverting buffers.

An arbiter enforces mutual-exclusion between two requests. It has internal analog circuitry to resolve a metastable state. Its behavior is described by the Petri net in figure 18. Whenever one or more input requests are pending (signalled by ***r1*** or ***r2***), ***the*** arbiter chooses to grant the resource to one of the requesters by signaling on the corresponding output ***g1*** or ***g2***. The user to whom the resource was granted can release it by signaling on ***d1*** or ***d2*** (“done”).

Merge elements (labelled ***M*** in the drawings) produce an output transition whenever they receive a new input

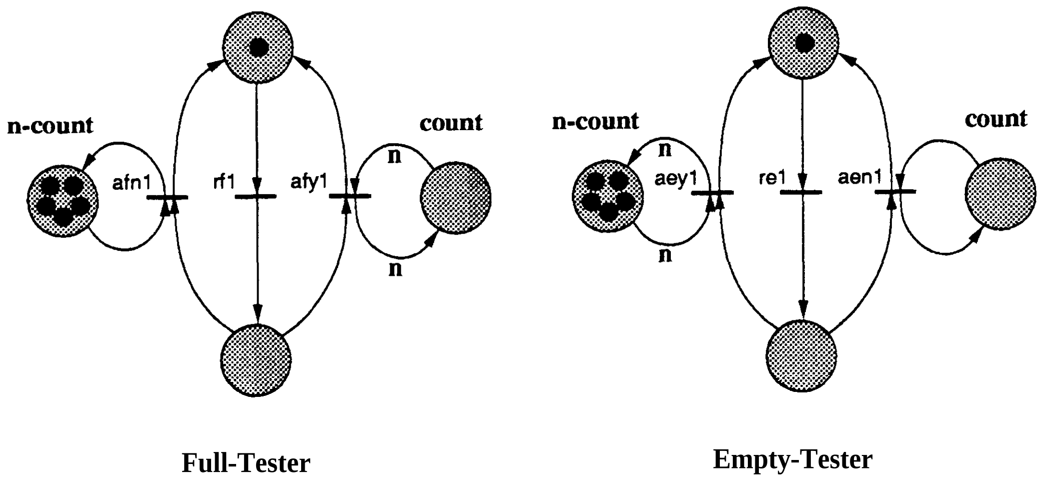

Figure 17: Petri nets for full and empty testers.

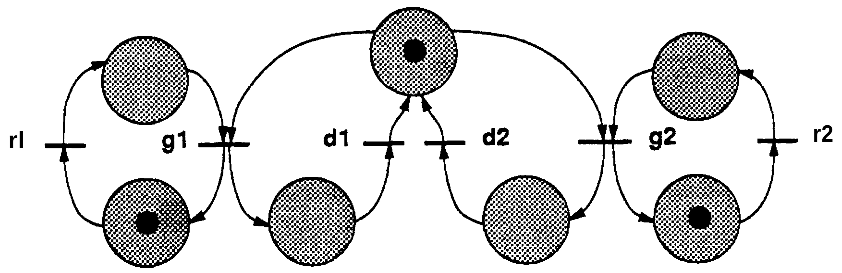

Figure 18 : Petri net for arbiter.

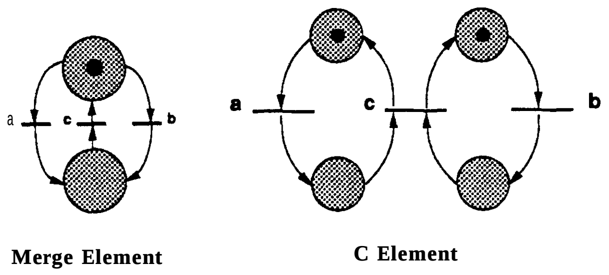

Figure 19: Petri nets for Merge and C elements.

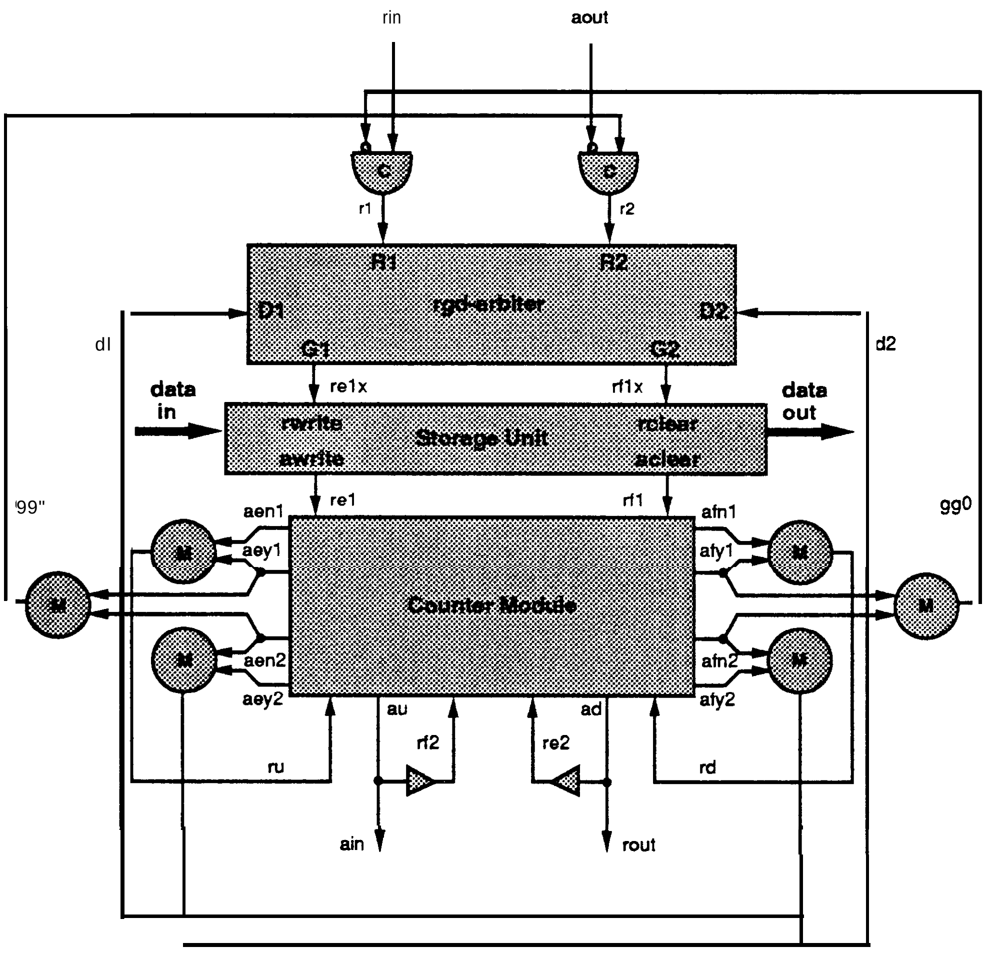

Figure 20: Controller with counter unit and storage unit.

transition. No two input transitions may occur unless separated by an output transition. In fact, Merge elements are simply XOR gates by another name.

The C ***element*** is an important primitive in speed-independent circuit design. A C element holds its output until both inputs have the logical value that is the complement of the output; it then changes the output to be the same as the inputs. C elements are often used to wait for two concurrent computations to complete. Petri nets for these two elements appear in figure 19.

4.2 Verification of the Circular Buffer

The verification strategy for this example is to consider the "implementation" to be the composition of the controller implementation and the storage unit (figure 20), where each component is described using the Petri nets above. The specification, as usual, will be the queue behavior. If the verification succeeds, we know that the queue behavior is met *and* that the storage unit is properly used. (Otherwise, verification would fail because of a failure in the storage unit; one component in the implementation would induce a failure in another component.)

We begin by verifying a one-queue — unsuccessfully. The verifier outputs the diagnostic trace:

rin *r1 re1x* ***re1 aey1*** *ru ggn r2 ain rin rf2 afy2 d1 rf1x rf1 afy1 gg0* ***r1 rd rout*** *re2 aey2* ***d2*** *re1x re1* ***aey1*** *ru ain.*

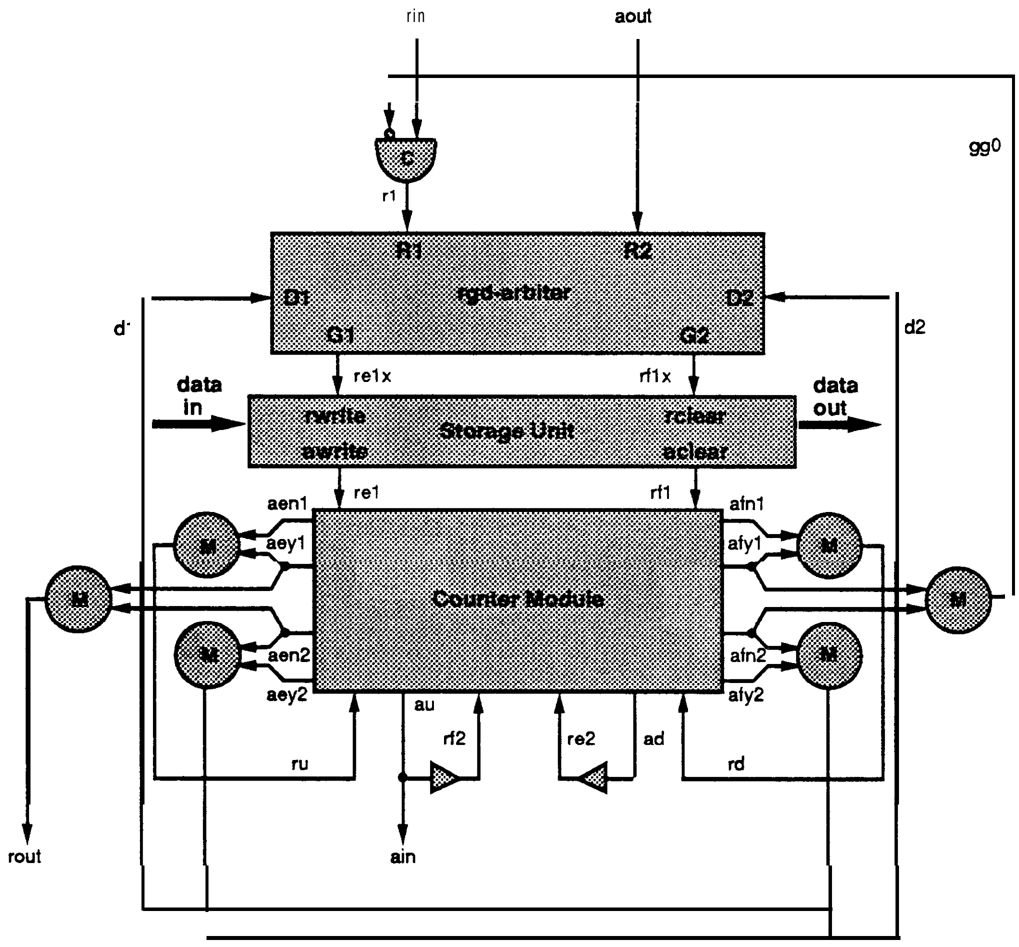

Figure 21: Correct controller implementation.

Deleting all internal wire names leaves the sequence ***rin ain rin rout ain***, which violates the specification — a one-queue ***is*** not ***allowed*** to signal ***ain*** twice without an ***aout*** between them. The problem is that, in this case, the controller allows a second piece of data to be added to the queue (***ain*** is signalled) before anything has been read from the memory (***aout*** has not arrived)[5].

A corrected implementation of the circular buffer queue is shown in figure 21. The verifier confirms that this corrected implementation of a one-queue conforms to a general one-queue specification.

As a final note, this example is one case where the explicit modelling of data would have been useful. The bug in the queue controller is in fact a misunderstanding between the controller and the storage unit about when data is valid (the controller correctly protects the storage unit from underflow and overflow).

Although ***we*** do not describe it in detail here, ***data validity*** can be explicitly modeled using trace structures without having to model the actual data ***values*** by representing a bundle of data wires by a single wire; transitions on this wire represent changes in the state of the ***validity*** of the data, rather than changes in the actual value. Hence, changes in data validity can be related to transitions on the control signals, since both transitions now appear in

[5]With some changes in the protocol of the storage unit, this controller can be used to implement a two-queue. Suppose the storage unit had an extra "'bin" to hold the currently available data output. In this case, *t-clear* **could mean a request to transfer data from the head of the RAM queue into this output bin (where it would be available for reading);** ***aclear*** **would acknowledge the transfer. After transfer, the old RAM storage cell would be available for reuse. This would be a valid implementation of a two-queue.**

traces.

Using this approach, bugs resulting from invalid data at the wrong time can be detected directly. In the circular buffer example above, the first failure was detected only after the queue appeared to be overfilled. However, the storage unit in fact was *not* overfilled. The real problem occurred earlier valid data was being "removed" from the unit before an ***rout*** was signalled to indicate that it was available. Thus the storage unit could accept new data, and the queue eventually appeared to overflow.

We have tried the same example, but modelling data validity. In this case, Regardless of the size of the queue implementation, the verifier detects that the first ***rout*** occurs when there is no valid output data in the storage unit; the data has already been removed, because the protocols are skewed.

5 Conclusions.

We set out to illustrate some thoughts about appropriate formalisms modelling, specifying, and verifying speed-independent circuits, using self-timed queues as examples.

We have found that as a user language, Petri nets are useful for specifying speed-independent circuits. They can sometimes also be used to model data computations.

Trace theory provides the simple compositional semantics for speed-independent circuits. In fact, trace theory provides the formal semantics for Petri nets as well. This is sometimes superior to the conventional notion that Petri nets are, of themselves, a formal semantics for concurrency, since we can choose between different Petri nets that all have the same underlying semantics.

We then introduced the conformation relation to formalize the notion of a circuit implementing a specification. This relation is a partial order, so that an implementation can exceed the minimum requirements of a specification. Formally, an implementation conforms to a specification when it can be safely substituted into any context in which the specification would function properly.

The use of the same formalism for modelling and specification makes hierarchical verification possible. Hierarchical verification can substantially reduce the state space explored during verification.

We have shown that automatic verification is a useful tool for finding obscure problems in tricky circuit designs, and that it can also aid in the development of good specifications.

There are limits to the power of trace theory, however. Trace theory cannot express liveness properties, so deadlocks may go undetected even in circuits that have been verified. Also, it is often not very suitable for modelling or specifying computations on data.

In the future, we hope to learn more by the consideration of examples. In addition, we intend to address some of the shortcomings of trace theory revealed above, including the inability to express liveness properties and data computations. Additionally, we are exploring alternative user-level languages for expressing trace structures, modelling and specification of real-time constraints, and the application of some of the ideas above to other domains than speed-independent circuits.

Acknowledgement

We would like to thank Ivan Sutherland for providing us with the circular buffer design.

References

[1] Tam-Anh Chu. On the Models for Designing VLSI Asynchronous Digital Systems. ***INTEGRATION, the VLSI journal,*** (4):99–1 13, 1986.

[2] David L. Dill. ***Trace Theory for Automatic Hierarchical Verification of Speed-independent Circuits.*** MIT Press, 1989.

[3] David L. Dill. Trace Theory for Automatic Hierarchical Verification of Speed-independent Circuits. In Jonathan Allen and F. Thomson Leighton, editor, ***Advanced Research in VLSI: Proceedings of the Fifth MIT Conference,*** MIT Press, 1988.

[4] D.L. Dill and E.M. Clarke. Automatic Verification of Asynchronous Circuits Using Temporal Logic. ***IEE Proceedings, Pt. E,*** 133(5):276–282, September 1986.

[5] M. Gordon. HOL: A Proof Generating System for Higher-Order Logic. In G. Birwistle and P. A. Subrahmanyam, editors, ***VLSI Speci'cation, Verification and Synthesis,*** Kluwer Academic Publishers, 1988.

[6] C.A.R. Hoare. ***A Model for Communicating Sequential Processes.*** Technical Report PRG-22, Programming Research Group, Oxford University Computing Laboratory, 1981.

[7] John E. Hopcroft and Jeffrey D. Ullman. ***Introduction to Automata Theory, Languages, and Computation.*** Addison-Wesley Publising Company, 1979.

[8] W. A. Hunt, Jr. The Mechanical Verification of a Microprocessor Design. In D. Borrione, editor, ***From HDL Descriptions to Guaranteed Correct Circuit Designs,*** North Holland, 1987.

[9] Teresa H.-Y. Meng, Robert W. Brodersen, and David G. Messerschmitt. Automatic Synthesis of Asynchronous Circuits from High-level Specifications. September 1988. Unpublished manuscript.

[10] Charles E. Molnar, Ting-Pien Fang, and Frederick U. Rosenberger. Synthesis of Delay-Insensitive Modules. ***In*** Henry Fuchs, editor, ***1985 Chapel Hill Conference on Very Large Scale Integration,*** pages 67-86, Computer Science Press, Inc., 1985.

[1 1] Suhas ***S.*** Patil. ***An Asynchronous Logic Array.*** Technical Report Technical Memorandom 62, Massachusetts Institute of Technology, Project MAC, 1975.

[12] James Lyle Peterson. ***Petri Net Theory and the Modeling of Systems.*** Prentice-Hall, Inc., 1981.

[13] Martin Rem, Jan L.A. van de Snepscheut, and Jan Tijmen Udding. Trace Theory and the Definition of Hierarchical Components. In Randal Bryant, editor, ***Third CalTech Conference on Very Large Scale Integration,*** pages 225-239, Computer Science Press, Inc., 1983.

[14] Charles L. Seitz. Ideas About Arbiters. ***Lambda,*** 10–14, First Quarter 1980.

[15] Ivan E. Sutherland. Micropipelines. ***Communications of the ACM,*** 32(6):720–738, June 1989.

[16] Jan Tijmen Udding. A Formal Model for Defining and Classifying Delay-insensitive Circuits and Systems. ***Distributed Computing,*** l(4): 197-204, 1986.

[17] Jan Tijmen Udding. ***Classification and Composition of Delay-Insensitive Circuits.*** PhD thesis, Department of Computing Science, Eindhoven University of Technology, September 1984.