Mechanically Verifying Safety and Liveness Properties of Delay Insensitive Circuits

David M. Goldschlag

Technical Report 66 March, 1991

Computational Logic Inc.
1717 W. 6th St. Suite 290
Austin, Texas 78703
(512) 322-9951

This work was supported in part at Computational Logic, Inc., by the Defense Advanced Research Projects Agency, ARPA Order 7406, and ONR Contract N00014-88-C-0454. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of Computational Logic, Inc., the Defense Advanced Research Projects Agency, the Office of Naval Research, or the U.S. Government.

Abstract

This paper describes, by means of an example, how one may mechanically verify delay insensitive circuits on an automated theorem prover. It presents the verification of both the safety and liveness properties of an n-node delay insensitive fifo circuit [12]. The proof system used is a mechanized implementation of Unity [3] on the Boyer-Moore prover [1], described in [6, 7, 8].

This paper describes the circuit formally in the Boyer-Moore logic and presents the mechanically verified correctness theorems. The formal description also captures the protocol that the circuit expects its environment to obey and specifies a class of suitable initial states.

This paper demonstrates how a general purpose automated proof system for concurrent programs may be used to mechanically verify both the safety and liveness properties of arbitrary sized delay insensitive circuits.

1. Introduction

General purpose theorem provers may be used to verify both safety and liveness properties of delay insensitive circuits. Although such mechanized proofs are not automatic, correctness properties may be both non-propositional and describe circuits of arbitrary size. Mechanical verification increases the trustworthiness of a proof. This paper describes the verification of an n-node first in first out (FIFO) queue.

The proof system used here is a version of a mechanized implementation of Unity [3, 6, 7, 8] on the Boyer-Moore prover [1]. The Unity logic is suitable for reasoning about programs under the interleaved model of concurrency. In this model, statements in a program run sequentially, but in an unknown order. Correctness properties are true only if they hold for all possible orderings.

Many researchers have used the interleaved model of concurrency as a basis for modeling delay insensitive circuits [11, 14, 3]. By restricting the power of program statements and assuming a non-deterministic yet weakly fair scheduling paradigm, interleaving adequately models circuit behavior. Martin's production rules [11] are statements in a non-deterministic program, and are obtained by correct refinements from higher level specifications. Programs in the Synchronized Transitions [14, 15] notation are similar to Unity programs and have been mechanically verified using LP [5]. However, Synchronized Transitions only provides for the verification of invariance properties.

This paper does not propose criteria for determining whether a circuit is truly delay insensitive. Rather, given a delay insensitive circuit, it describes how to verify its correctness properties under the interleaved model of concurrency. The example here formalizes an n-node (FIFO) queue and presents the verification of its safety and the liveness properties. The basic element in this circuit is described in [12].

This paper is organized in the following way. Section 2 briefly describes the Boyer-Moore logic, its prover, and the mechanized implementation of Unity. Section 3 defines the FIFO circuit. Section 4 presents the correctness theorems, which are proved in section 5. Section 6 discusses related work and offers concluding remarks.

2. The Proof System

Mechanized Unity is implemented in the Nqthm version of the Boyer-Moore logic [1] enhanced with a facility for defining fully quantified definitions [10] and the Kaufmann proof checker [9]. Nqthm is a quantifier-free first order logic with a prefix syntax and semantics similar to pure Lisp.

Mechanized Unity defines most of its specification predicates with respect to an operational semantics characterizing an arbitrary fair execution of a concurrent program. A fair execution is an infinite sequence of states, obtained from some initial state by the sequential application of program statements. The only restriction on the scheduling of statements is weak fairness: every statement must be scheduled infinitely often. Since specifications are defined with respect to an arbitrary fair execution, proved specifications are true for all fair executions. Consequently, specifications neither assume nor guarantee any particular timing characteristics of the program.

Specifications in this logic are proved by the use of proof rules that have been adapted from the Unity logic. These proof rules permit concise non-operational correctness proofs. The proof rules are sound because they are theorems of the operational semantics of concurrency described earlier.

We now illustrate terms in the logic by introducing the proof system's specification predicates. Assuming the term **`(INITIAL-CONDITION IC PRG)`** implies that **`IC`** holds on the initial state of any execution of the program **PRG**. In a similar way, the predicates for safety and liveness properties are defined:

- **(UNLESS P Q PRG)** means that, for all statements **S** in program **PRG**, **P∧¬Q {S} P∨Q**. This means that **P** persists until **Q** holds. In particular, the term **(UNLESS P '(FALSE) PRG)** implies that once **P** holds in an execution of **PRG**, it continues to hold, since every statement preserves **P**. **UNLESS** terms of this sort are called stability properties, since the predicate, once reached, is stable.
- If a stable predicate is true on the initial state, then that predicate is also an invariant, since it holds throughout the execution. The term **(INVARIANT P PRG)** means that **P** holds on every state in the execution of **PRG**. To prove this, one may have to make assumptions about the initial state.
- The term **(LEADS-TO P Q PRG)** means that every **P** state (in an execution of **PRG**) is eventually followed by a **Q** state (**Q** may hold true immediately).

The following functions are also used in this paper:

- **T** is an abbreviation for **(TRUE)** which is not equal to **F** which is an abbreviation for **(FALSE)**.
- **(EQUAL A B)** is **T** if **A=B**, **F** otherwise.
- The value of the term **(AND X Y)** is **T** if both **X** and **Y** are not **F**, **F** otherwise. **OR**, **IMPLIES**, **NOT**, and **IFF** are similarly defined.
- The value of the term **(IF A B C)** is **C** if **A=F**, **B** otherwise.
- **(NUMBERP A)** tests whether **A** is a number.
- **(ZEROP A)** is **T** if **A=0** or **(NOT (NUMBERP A))**.
- **(ADD1 A)** returns the successor to **A** (i.e., **A+1**). If **(NUMBERP A)** is **FALSE** then **(ADD1 A)** is **1**.
- **(SUB1 A)** returns the predecessor of **A** (i.e., **A-1**). If **(ZEROP A)** is **TRUE**, then **(SUB1 A)** is **0**.
- **(LESSP A B)** is **A<B**, and is defined recursively using **SUB1**.
- Literals are quoted. For example, **'ABC** is a literal. **NIL** is an abbreviation for **'NIL**.
- **(CONS A B)** represents a pair.
- **(LISTP A)** is **TRUE** if **A** is a pair (i.e., **A** is a non-empty list).
- **(LIST A)** is an abbreviation for **(CONS A NIL)**. **LIST** can take an arbitrary number of arguments: **(LIST A B C)** is read as **(CONS A (CONS B (CONS C NIL)))**.
- **'(A)** is an abbreviation for **(LIST 'A)**. Similarly, **'(A B C)** is an abbreviation for **(LIST 'A 'B 'C)**.[1]
- **(LENGTH L)** returns the length of the list **L**.

Recursive definitions are permitted, provided termination can be proved. For example, the function **APPEND**, which appends two lists, is defined as:

Definition: Append

```
(APPEND X Y)
   =
(IF (LISTP X)
    (CONS (CAR X) (APPEND (CDR X) Y))
    Y)
```

[1] Actually, this quote mechanism is a facility of the Lisp reader [16].

This function terminates because the measure **`(LENGTH X)`** decreases in each recursive call.

A program in Mechanized Unity is a list of statements, where each statement has the form **`'(LIST FUNCTION-NAME ARG-1`** ... **`ARG-N)`**. The arguments may be wire names, if one wishes to use the same function in several statements. For example, the statement representing a NOR gate may be:

```
(LIST 'NOR-GATE A B C)
```

where **`A`** and **`B`** are understood to be input wires and **`C`** is the output wire. The literal **`'NOR-GATE`** refers to the following function:

Definition: Nor-Gate

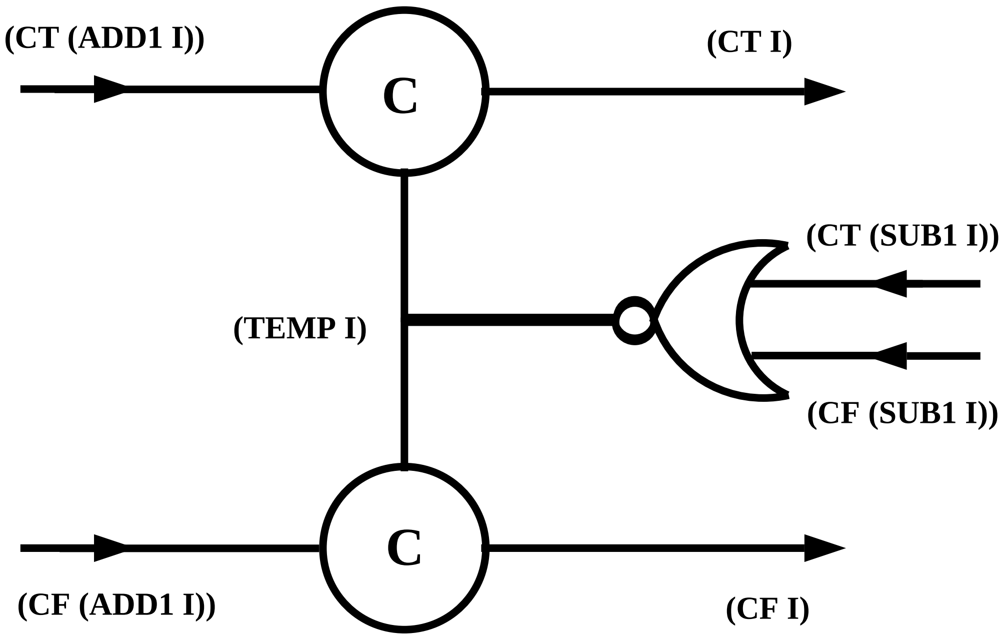
```
(NOR-GATE OLD NEW A B C)
   =
(AND (IFF (VALUE NEW C)
          (NOT (OR (VALUE OLD A)
                   (VALUE OLD B))))
     (CHANGED OLD NEW (LIST C)))
```

Each function implementing a statement takes two arguments in addition to the ones specified in the statement. These two arguments represent the states before and after the execution of the statement. The function returns **`TRUE`** only if the **`NEW`** state is a possible successor state to the **`OLD`** state. In this case, the output wire must be the **`NOR`** of the input wires. (This function will be discussed completely in the next section.) It is useful for statements to be defined by functions that have access to both the previous and next states, since this permits non-deterministic transitions. Non-determinism simplifies modeling a circuit's environment, for example.

3. The FIFO Circuit

This FIFO circuit is composed of a producer and a consumer which *push* values upon and *pop* values from the internal nodes of the queue. The internal nodes are a sequence of similar nodes, each differing from the other by an index. Each node contains at most one bit; it may be **`TRUE`**, **`FALSE`**, or empty. A node attains its predecessor's value once it determines that its value has been copied to its successor. A node does not become empty simply because its value is copied to its successor. Therefore, in order for this circuit to operate correctly, the producer must push an empty value upon the queue between pushes of non-empty values. Furthermore, a popped value is considered non-empty only if it is non-empty and the previous popped value was empty. Intuitively, a value propagates along the queue leaving a trail of identical values. These copies are cleaned up by the empty value that is pushed upon the queue to delimit the next non-empty value.

An **`N`** node queue has **`N-2`** internal nodes, indexed **`N-1,`** ... **`, 1`**. The **`I`**'th internal node in the FIFO circuit has the following components:



The labels on the wires are wire names; notice that each of the output wires from the C-elements [13] actually fork; one branch connects to the input of the successor node's corresponding C-element; the other connects to the predecessor's NOR gate. We take these forks to be isochronic [11] (assume that the signal propagates simultaneously to the gates at the end of each fork).

Each node behaves in the following way: A bit is encoded by double-rail coding. **`TRUE`** is represented by the C-element [13] **`CT`** being **`TRUE`**, and the other C-element **`CF`** being **`FALSE`**. **`FALSE`** is represented by the opposite configuration. If the node is empty, both C-elements are **`FALSE`**; never will both C-elements be **`TRUE`** simultaneously. This is because this circuit requires (and maintains) that if two adjacent nodes are non-empty, they must also represent the same value.

A node copies a new value from its predecessor when its successor differs from its predecessor. For example, assume that the successor is empty, and the predecessor is non-empty. Therefore, the incoming **`TEMP`** becomes **`TRUE`** and permits the other C-elements to become true, if their other inputs are **`TRUE`**.

In Mechanized Unity, this node is described by three statements corresponding to the two C-elements and single NOR gate components. The NOR gate is represented by the following function:

Definition: Nor-Gate

```
(NOR-GATE OLD NEW A B C)
   =
(AND (IFF (VALUE NEW C)
          (NOT (OR (VALUE OLD A)
                   (VALUE OLD B))))
     (CHANGED OLD NEW (LIST C)))
```

The term **`(VALUE OLD A)`** looks up the value of the variable named **`A`** in state **`OLD`**. **`(CHANGED OLD NEW (LIST C))`** states that only the variable **`C`** may change between states **`OLD`** and **`NEW`**. This function says that the value of **`C`** in state **`NEW`** becomes the *nor* of the values of **`A`** and **`B`** in state **`OLD`**. **`OLD`** and **`NEW`** represent successive states in the execution of the program. The statement for the NOR gate in the **`I`**'th node of the queue must instantiate **`A`**, **`B`**, and **`C`** to be the appropriate wire names. The statement is:

```
(LIST 'NOR-GATE  (CT (SUB1 I)) (CF (SUB1 I)) (TEMP I))
```

A statement is a list; the first element is the name of the function representing the VLSI component, remaining elements are the names of the input and output wires of that component. Since wire names are indexed, the functions **`CT`**, **`CF`** and **`TEMP`** take arguments.

Similarly, the C-element is described by the following function:

Definition: C-Element

```
(C-ELEMENT OLD NEW A B C)
   =
(IF (IFF (VALUE OLD A)
         (VALUE OLD B))
    (AND (IFF (VALUE NEW C) (VALUE OLD A))
         (CHANGED OLD NEW (LIST C)))
  (CHANGED OLD NEW NIL))
```

This function states that **`C`** in state **`NEW`** becomes equal to the inputs, if both inputs **`A`** and **`B`** are equivalent in state **`OLD`**; otherwise, all variables remain unchanged (**`NIL`** is the empty list). This function is used in the following two statements, each representing a single C-element:

```
(LIST 'C-ELEMENT (CT (ADD1 I)) (TEMP I) (CT I))
(LIST 'C-ELEMENT (CF (ADD1 I)) (TEMP I) (CF I))
```

A single node of the FIFO circuit is a collection of the two statements representing the two C-elements and the single statement representing the NOR gate. We define the function **`FIFO-NODE`** to collect the three statements in node **`I`**:

Definition: Fifo-Node

```
(FIFO-NODE I)
   =
(LIST (LIST 'C-ELEMENT (CT (ADD1 I)) (TEMP I) (CT I))
      (LIST 'C-ELEMENT (CF (ADD1 I)) (TEMP I) (CF I))
      (LIST 'NOR-GATE  (CT (SUB1 I)) (CF (SUB1 I)) (TEMP I)))
```

The fact that the **`TEMP`** wire is truly an isochronic fork is apparent in this formula. That is, **`(TEMP I)`** is the output of the NOR gate, and is an input to two C-elements. Adding a function that copies **`TEMP`** to another wire would add complexity to the verification without changing the overall behavior of the circuit. The output of each C-element is also an isochronic fork.

The internal nodes in our n-node queue will have indices **`(N-1, ..., 1)`**. Nodes **`N`** and **`0`** will be, respectively, producer and consumer nodes. These nodes must obey the four-phase signalling that this queue expects, and keep track of the *pushed* and *popped* values. The producer node is defined as follows:

Definition: In-Node

```
(IN-NODE OLD NEW I)
   =
(IF (IFF (VALUE OLD (TEMP I))
         (EMPTY-NODE OLD I))
    (IF (EMPTY-NODE OLD I)
        (OR (CHANGED OLD NEW NIL)
            (AND (OR (TRUE-NODE NEW I)
                     (FALSE-NODE NEW I))
                 (EQUAL (VALUE NEW 'INPUT)
                        (CONS (TRUE-NODE NEW I)
                              (VALUE OLD 'INPUT)))
                 (CHANGED OLD NEW (LIST (CT I) (CF I)
                                        'INPUT))))
      (AND (EMPTY-NODE NEW I)
           (CHANGED OLD NEW (LIST (CT I) (CF I)))))
  (CHANGED OLD NEW NIL))
```

The term **(EMPTY-NODE OLD I)** tests whether this **I**'th node is empty in state **OLD**. (Neither C-element in node **I** is **TRUE**.) Terms **(TRUE-NODE NEW I)** and **(FALSE-NODE NEW I)** test whether the **I**'th node in state **NEW** contains a **TRUE** or **FALSE** bit, respectively. In our example, the producer node will have index **N**. Its behavior is as follows: If the value of the tail of the queue (node **N**) has already been copied into node **N-1** (as indicated by the value of **(TEMP N)**) and the tail of the queue is empty, then a new value *may* be placed upon the tail of the queue. If the new value is **TRUE** or **FALSE**, then the variable **INPUT** is updated to reflect the newly pushed value. If the tail of the queue is not empty, yet has already been copied, then an empty value is placed upon the tail of the queue. If the tail of the queue has not yet been copied, no change occurs.

The producer node is non-deterministic, since it need not push a new value upon the queue unless it pushed a non-empty value earlier. That is, this statement may execute repeatedly without ever pushing a non-empty value.

The consumer node is defined as follows:

Definition: Out-Node

```
(OUT-NODE OLD NEW)
   =
(AND (IFF (VALUE NEW (CT 0))
          (VALUE OLD (CT 1)))
     (IFF (VALUE NEW (CF 0))
          (VALUE OLD (CF 1)))
     (IF (AND (EMPTY-NODE OLD 0)
              (NOT (EMPTY-NODE NEW 0)))
         (EQUAL (VALUE NEW 'OUTPUT)
                (CONS (TRUE-NODE NEW 0)
                      (VALUE OLD 'OUTPUT)))
       (EQUAL (VALUE NEW 'OUTPUT) (VALUE OLD 'OUTPUT)))
     (CHANGED OLD NEW (LIST (CT 0) (CF 0) 'OUTPUT)))
```

The consumer node's index is **0**. Node **1** is copied into the head of the queue. If the head of the queue is thereby changed from empty to non-empty, then the variable **OUTPUT** representing popped values is updated appropriately. Since the schedule of statements is unknown, the internal nodes in the queue cannot depend upon the rate at which values are popped.

The entire queue, consisting of a consumer, the internal nodes, and a producer (with the extra **(TEMP N)** line), is represented using the following three functions. The first collects the internal nodes:

Definition: Internal-Nodes

```
(INTERNAL-NODES N)
   =
(IF (ZEROP N)
    NIL
  (APPEND (FIFO-NODE N)
          (INTERNAL-NODES (SUB1 N))))
```

The next function collect the statements describing the external nodes:

Definition: External-Nodes

```
(EXTERNAL-NODES N)
   =
(LIST (LIST 'IN-NODE N)
      (LIST 'OUT-NODE)
      (LIST 'NOR-GATE  (CT (SUB1 N)) (CF (SUB1 N)) (TEMP N)))
```

Finally, the entire circuit is captured by the term **(FIFO-QUEUE N)**:

Definition: Fifo-Queue

```
(FIFO-QUEUE N)
   =
(APPEND (EXTERNAL-NODES N)
        (INTERNAL-NODES (SUB1 N)))
```

In the correctness specifications, we use the term **(FIFO-QUEUE N)** denoting a FIFO queue of length **N**. As with all variables, **N** is universally quantified, so the theorems are true for queues of any length. (A hypothesis in these theorems requires that **N** exceed **1**, implying the existence of at least one internal node.)

4. The Correctness Specifications

The important correctness properties, that pushed values are not lost, and that pushed values are eventually popped, both depend upon a invariant that characterizes legal states. Recall that the correct operation of the circuit depends upon adjacent non-empty nodes being equivalent. In addition, if a node differs from its successor, then its incoming **TEMP** wire must be up-to-date. These requirements are formalized in the following way:

Definition: Proper-Node

```
(PROPER-NODE STATE I)
   =
(AND (IMPLIES (AND (NOT (EMPTY-NODE STATE I))
                   (EMPTY-NODE STATE (SUB1 I)))
              (VALUE STATE (TEMP I)))
     (IMPLIES (AND (EMPTY-NODE STATE I)
                   (NOT (EMPTY-NODE STATE (SUB1 I))))
              (NOT (VALUE STATE (TEMP I))))
     (OR (TRUE-NODE STATE I)
         (FALSE-NODE STATE I)
         (EMPTY-NODE STATE I))
     (IMPLIES (NOT (EMPTY-NODE STATE I))
              (OR (EMPTY-NODE STATE (SUB1 I))
                  (IF (TRUE-NODE STATE I)
                      (TRUE-NODE STATE (SUB1 I))
                    (FALSE-NODE STATE (SUB1 I))))))
```

The term **(PROPER-NODES STATE N)** checks whether nodes **(N, ..., 1)** are proper. The invariance property is stated as follows:

Theorem: Proper-Nodes-Invariant

```
(IMPLIES (AND (LESSP 1 N)
              (INITIAL-CONDITION '(PROPER-NODES STATE
                                                (QUOTE ,N))
                                 (FIFO-QUEUE N)))
         (INVARIANT '(PROPER-NODES STATE (QUOTE ,N))
                    (FIFO-QUEUE N)))
```

This theorem states that if the initial state is legal, then all subsequent states are legal. The legal state predicate is encoded as a backquoted [16] term, in the following way: The first element of the term is the function symbol **PROPER-NODES**, so **PROPER-NODES** is the function that is invariant. The second element is **STATE**, which is a dummy literal: upon evaluating the backquoted term in the context of some state in the execution, **STATE** is bound to that state. The third element is **(QUOTE ,N)** which is a shorthand for introducing a variable into the formula. That is, the **N** in the hypothesis is the same **N** that is in the conclusion, and is the same universally quantified **N** specifying the size of the queue that we are describing via the function **(FIFO-QUEUE N)**.

The next invariant states that values are consumed in the order in which they are produced. To specify this, we define the term **(QUEUE-VALUES STATE N)** that returns a list of the values in the queue:

Definition: Queue-Values

```
(QUEUE-VALUES STATE N)
   =
(IF (ZEROP N)
    NIL
  (IF (AND (NOT (EMPTY-NODE STATE N))
           (EMPTY-NODE STATE (SUB1 N)))
      (CONS (TRUE-NODE STATE N)
            (QUEUE-VALUES STATE (SUB1 N)))
    (QUEUE-VALUES STATE (SUB1 N))))
```

Specifically, a node only *counts* if it is non-empty and its successor is empty. The invariant depends upon the queue being in a legal configuration and is specified in the following way:

Theorem: Queue-Values-Invariant

```
(IMPLIES (AND (INITIAL-CONDITION
               '(AND (PROPER-NODES STATE (QUOTE ,N))
                     (EQUAL (VALUE STATE (QUOTE INPUT))
                            (APPEND (QUEUE-VALUES STATE
                                                  (QUOTE ,N))
                                    (VALUE STATE
                                           (QUOTE OUTPUT)))))
               (FIFO-QUEUE N))
              (LESSP 1 N))
         (INVARIANT '(EQUAL (VALUE STATE (QUOTE INPUT))
                            (APPEND (QUEUE-VALUES STATE
                                                  (QUOTE ,N))
                                    (VALUE STATE
                                           (QUOTE OUTPUT))))
                    (FIFO-QUEUE N)))
```

This invariant states that the produced values always equal the concatenation of the values in the queue and the consumed values. Interestingly, this invariant can be satisfied by an incorrect program: we must also prove that the variables **INPUT** and **OUTPUT** only grow. These statements have been proved as **UNLESS** properties stating that for all statements in the program, the variables **INPUT** and **OUTPUT** either remain unchanged, or become extended by the values **TRUE** or **FALSE**.

Theorem: Input-Only-Adds-Boolean

```
(IMPLIES (LESSP 1 N)
         (UNLESS '(EQUAL (VALUE STATE (QUOTE INPUT))
                         (QUOTE ,K))
                 '(OR (EQUAL (VALUE STATE (QUOTE INPUT))
                             (CONS (TRUE) (QUOTE ,K)))
                      (EQUAL (VALUE STATE (QUOTE INPUT))
                             (CONS (FALSE) (QUOTE ,K))))
                 (FIFO-QUEUE N)))
```

Theorem: Output-Only-Adds-Boolean

```
(IMPLIES (LESSP 1 N)
         (UNLESS '(EQUAL (VALUE STATE (QUOTE OUTPUT))
                         (QUOTE ,K))
                 '(OR (EQUAL (VALUE STATE (QUOTE OUTPUT))
                             (CONS (TRUE) (QUOTE ,K)))
                      (EQUAL (VALUE STATE (QUOTE OUTPUT))
                             (CONS (FALSE) (QUOTE ,K))))
                 (FIFO-QUEUE N)))
```

The liveness condition requires that values be passed through the queue. Without tagging queue values, this must be stated in the following way: if the queue is non-empty, then eventually the number of consumed values increases. This is expressed in the following `LEADS-TO` property:

Theorem: Output-Grows

```
(IMPLIES (AND (INITIAL-CONDITION '(PROPER-NODES STATE N)
                                 (FIFO-QUEUE N))
              (LESSP 1 N))
         (LEADS-TO '(AND (LISTP (QUEUE-VALUES STATE N))
                         (EQUAL (LENGTH (VALUE STATE
                                               (QUOTE OUTPUT)))
                                (QUOTE ,K)))
                   '(LESSP (QUOTE ,K)
                           (LENGTH (VALUE STATE
                                          (QUOTE OUTPUT))))
                   (FIFO-QUEUE N)))
```

These correctness properties have been mechanically verified on the Boyer-Moore prover, using many intermediate theorems.

5. The Correctness Proof

The proof of the invariance theorems proceeded by case analysis on the various statements in the program. Since the functions specifying both legal states and queue values are defined recursively, the proofs of these theorems were inductive and required that generalizations of the invariance theorems be proved first. It is unfortunate that the legal state invariant cannot be decomposed: although, the invariant is really three conjuncts, the stability of each depends upon all three.

The liveness property is a more interesting proof and is the focus of this section. We wish to prove that non-empty values on the queue are eventually popped off the queue; this was formalized by stating that the length of the history variable `OUTPUT` recording popped values eventually increases. We prove this by demonstrating a decreasing measure: non-empty values move forward in the queue; when one reaches node `1`, it is popped and the length of `OUTPUT` grows. We prove the decreasing measure in a restricted sense: if a queue value is non-empty and the entire subqueue ahead of it is empty, then that queue value moves forward. It is obvious that any non-empty queue also has a most forward element, so it is sufficient to prove this theorem. Furthermore, it is simpler to prove this theorem than to consider the interactions of unknown elements in the queue. The theorem is stated in the following way:

Theorem: Full-Rest-Empty-Moves-Forward

```
(IMPLIES (AND (INITIAL-CONDITION '(PROPER-NODES STATE
                                                 (QUOTE ,N))
                                  (FIFO-QUEUE N))
              (LESSP 1 N)
              (LESSP I N)
              (NOT (ZEROP I)))
         (LEADS-TO '(AND (NOT (EMPTY-NODE
                                 STATE (QUOTE ,(ADD1 I))))
                         (AND (EMPTY-NODE STATE (QUOTE ,I))
                              (NOT (LISTP (QUEUE-VALUES
                                             STATE
                                             (QUOTE ,I))))))
                   '(AND (NOT (EMPTY-NODE STATE (QUOTE ,I)))
                         (AND (EMPTY-NODE STATE
                                            (QUOTE ,(SUB1 I)))
                              (NOT (LISTP
                                      (QUEUE-VALUES
                                       STATE
                                       (QUOTE ,(SUB1 I)))))))
                   (FIFO-QUEUE N)))
```

This theorem can be applied inductively, since it says that if the **`(ADD1 I)`**'th queue value is the topmost non-empty value, eventually the **`I`**'th element will be the topmost non-empty value. This is applied inductively until the node **`1`** is the topmost non-empty value. The following theorem states that **`OUTPUT`** then grows:

Theorem: Output-Grows-Immediately

```
(IMPLIES (AND (INITIAL-CONDITION '(PROPER-NODES STATE
                                                 (QUOTE ,N))
                                  (FIFO-QUEUE N))
              (LESSP 1 N))
         (LEADS-TO '(AND (NOT (EMPTY-NODE STATE 1))
                         (AND (EMPTY-NODE STATE 0)
                              (EQUAL (LENGTH (VALUE STATE
                                                     'OUTPUT))
                                     (QUOTE ,K))))
                   '(LESSP (QUOTE ,K)
                           (LENGTH (VALUE STATE 'OUTPUT)))
                   (FIFO-QUEUE N)))
```

The theorem **Output-Only-Adds-Boolean** implies that the length of **`OUTPUT`** only grows. This fact, along with the previously stated two theorems, permits the proof of the liveness property, **Output-Grows**.

6. Conclusion

This paper demonstrates how techniques for reasoning about concurrent programs may be applied to delay insensitive circuits. In this case, a proof system mechanizing Unity has been used to specify and verify both safety and liveness properties for an n-node FIFO circuit. This specification of the queue formalizes the assumptions about its environment. Mechanized Unity permits the mechanically verified proof of circuits of arbitrary size.

This work is similar to Synchronized Transitions [14], especially the later work [15] which was mechanized on the Larch Prover [5]. Synchronized Transitions uses a syntax similar to Unity for specifying hardware. It can only be used, however, to prove invariance properties. The invariance property of a high level specification of a FIFO circuit was mechanically verified in [15]. Synchronized Transitions does provide a nice composition mechanism for hierarchical circuit design.

Other research has produced promising techniques for fully automatic verification of certain safety [4] and liveness properties [2] using trace theory and model checking. These systems check whether a finite state machine satisfies a formula by, essentially, completely simulating the machine. If the machine does not satisfy the formula, the system can return an offending trace; this facility is useful for debugging. Such systems may be more useful than semi-automatic techniques for verifying fixed size circuit components, since invariants specifying legal states become very complicated. However, these systems cannot reason about arbitrary sized components or about non-propositional correctness properties. Also one must still determine the circuit's suitable initial states, which, in the general case, is similar to determining invariants. A useful system (which does not yet exist) might combine automatic techniques for verifying fixed sized circuit components with semi-automatic techniques for combining these components.

There are two assumptions underlying this work. The first is that the behavior of delay insensitive circuits is accurately modeled by the interleaved model of concurrency. This assumption permits one to ignore isochronic forks and use the same wire in several inputs. The second is that the circuit being verified is truly delay insensitive. Several criteria have been proposed to test delay insensitivity: Martin checks whether his production rules map to VLSI components and the rules' preconditions are mutually exclusive. Straunstrup et. al, propose the two conditions of consumed values and correspondence, while Chandy and Misra suggest stability of preconditions. Since these conditions sometimes conflict, characterizing delay insensitive circuits remains an incompletely answered question. In this paper, an arbitrary sized circuit known to be delay insensitive was verified under the interleaved model of concurrency.

References

1. R. S. Boyer and J S. Moore. *A Computational Logic Handbook.* Academic Press, Boston, 1988.

2. Jerry R. Burch. Combining CTL, Trace Theory, and Timing Models. In *Automatic Verification Methods for Finite State Systems*, J. Sifakis, Eds., Springer-Verlag, 1990, pp. 334-348.

3. K. Mani Chandy and Jayadev Misra. *Parallel Program Design: A Foundation.* Addison Wesley, Massachusetts, 1988.

4. David L. Dill. *Trace Theory for Automatic Hierarchical Verification of Speed-Independent Circuits.* The MIT Press, Cambridge, Massachusetts, 1988.

5. S.J. Garland, J.V. Guttag, J.J. Horning. "Debugging Larch Shared Language Specifications". *IEEE Transactions on Software Engineering SE-16*, 9 (September 1990).

6. David M. Goldschlag. Mechanizing Unity. In *Programming Concepts and Methods*, M. Broy and C. B. Jones, Eds., North Holland, Amsterdam, 1990.

7. David M. Goldschlag. "Mechanically Verifying Concurrent Programs with the Boyer-Moore Prover". *IEEE Transactions on Software Engineering SE-16*, 9 (September 1990).

8. David Goldschlag. A Mechanical Formalization of Several Fairness Notions. Tech. Rept. 65, Computational Logic, Inc. Austin Texas 78703, March, 1991.

9. M. Kaufmann. A User's Manual for an Interactive Enhancement to the Boyer-Moore Theorem Prover. Tech. Rept. ICSCA-CMP-60, Institute for Computing Science, University of Texas at Austin, Austin, TX 78712, 1987. Also available through Computational Logic, Inc., Suite 290, 1717 West Sixth Street, Austin, TX 78703.

10. Matt Kaufmann. DEFN-SK: An Extension of the Boyer-Moore Theorem Prover to Handle First-Order Quantifiers. Tech. Rept. 43, Computational Logic, Inc., May, 1989. Draft.

11. Alain J. Martin. "Compiling Communicating Processes into Delay-Insensitive VLSI Circuits". *Distributed Computing 1* (1986), 226-234.

12. Alain J. Martin. Self-Timed FIFO: An Exercise in Compiling Programs into VLSI Circuits. In *From HDL Descriptions to Guaranteed Correct Circuit Designs*, North-Holland, Amsterdam, 1987, pp. 133-153.

13. R. E. Miller. *Switching Theory.* Wiley, 1965.

14. J. Staunstrup and M.R. Greenstreet. Designing Delay Insensitive Circuits using ‘‘Synchronized Transitions’’. In *Proceedings of the IMEC-IFIP International Workshop on Applied Formal Methods for Correct VLSI Design*, Dr. Luc Claesen, Eds., Elsevier Science Publishers B.V., Amsterdam, 1989, pp. 741-758.

15. Jorgen Staunstrup, Stephen J. Garland, and John V. Guttag. Localized Verification of Circuit Descriptions. In *Automatic Verification Methods for Finite State Systems*, J. Sifakis, Eds., Springer-Verlag, 1990, pp. 348-364.

16. G. L. Steele, Jr. *Common Lisp The Language.* Digital Press, 30 North Avenue, Burlington, MA 01803, 1984.

Table of Contents