Using `Synchronized Transitions` for Simulation and Timing Verification

Mark R. Greenstreet
Department of Computer Science
Princeton University
Princeton, NJ 08544

Abstract

`Synchronized Transitions` is a formal notation for hardware specification, verification, and simulation. This paper describes the use of `Synchronized Transitions` in the design of a chip for high bandwidth interprocessor communication. The chip uses a hybrid of synchronous and self-timed circuit techniques; a proof is presented that all timing requirements are satisfied. The `Synchronized Transitions` notation is presented, and it is shown how programs can be translated into logic predicates, providing a basis for formal verification. The use of `Synchronized Transitions` in the simulation of the chip is described, and the design choices of using both simulation and formal proofs are discussed.

Keywords: Formal hardware verification, invariants, real-time systems, self-timed circuits, simulation, timing verification.

1 Introduction

Formal methods can be used for the specification, verification, simulation, and synthesis of digital hardware. A formal method consists of a notation for writing expressions, and rules for manipulating these expressions. `Synchronized Transitions` [RS87] is a notation in which digital circuits are modeled as parallel programs. These programs can be transformed into predicates that specify the state transitions that are allowed by the program. With this translation, predicate calculus can be used to derive rules for manipulating expressions. Other transformations can be applied to the notation to produce simulation programs or to synthesize hardware.

This paper presents the application of `Synchronized Transitions` to the design of a full-custom chip. The primary motivation for designing this chip is to demonstrate a novel approach to interprocessor communication [GL90]; formal methods have been employed where they offered a reduction in design time or a significant increase in the likelihood of a correct design. In this approach, formal proofs are used to verify circuits that were either novel or complicated. Simulation is used extensively to informally check many aspects of the design. `Synchronized Transitions` is used for both simulation and formal verification; examples of these complementary applications of the same notation are presented throughout this paper.

In this paper, section 2 presents the `Synchronized Transitions` language; showing how programs can be translated into logic formulas for formal verification, or executed for simulation. Section 3 describes the multiprocessor and communication chip. A proof for the timing correctness of the chip is sketched in section 4; the complete program and proof are given in the appendices. This proof differs from many hardware related proofs in that it addresses timing issues. The remainder of the paper discusses simulation of the design and the trade-offs that motivate using both simulation and formal verification in the design process.

2 Synchronized Transitions

2.1 Transitions and combinators

A `Synchronized Transitions` program is a collection of state variables and a **transition valued expression**. This expression defines the set of allowed state transitions. An execution is a possibly infinite sequence of states.

The simplest expression is a **transition**. A transition is composed of a **precondition** and an **action**. When the precondition is satisfied, the action may be performed. Syntactically, transitions are written with the form:

$\ll$ *precondition* $\rightarrow$ *action* $\gg$

The precondition is a Boolean valued expression and the action is a multiassignment. To avoid conflicting assignments, the variables appearing on the left side of the multiassignment must be disjoint. For example

$\ll$ `a < b` $\rightarrow$ `a, b := b, a` $\gg$

is a transition that exchanges `a` and `b` if they are not in descending order.

Three syntactical abbreviations for transitions are supported.

$\ll$ *action* $\gg$
This is a transition whose precondition is always satisfied. For example,
$\ll$ `a := b` $\gg$ $\equiv$ $\ll$ `TRUE` $\rightarrow$ `a := b` $\gg$

$\ll$ *precondition* $\gg$
This is a transition that has a precondition but performs no action. This form is useful in combination with the combinators described below.

$\ll\gg$ $\equiv$ $\ll$ `TRUE` $\gg$
This is the combination of the two preceding abbreviations.

Transition valued expressions are defined by two productions:

1. A transition is a transition valued expression.

2. $t_1 \circ t_2$ is a transition valued expression, where t_1 and t_2 are transition valued expressions, and $\circ$ is one of $\|$, $*$, and $+$ which are described below.

The **asynchronous combinator**, $\|$, specifies that two expressions are executed independently; there is no global thread of control determining the order of execution. If either precondition is satisfied, the corresponding action may be performed. If both preconditions are satisfied, one of the transitions is selected *non-deterministically.* Furthermore, transitions combined with $\|$ are executed atomically; the sequence of states of any execution corresponds to some sequential execution of transitions[1]. For example

$\ll$ `a < b` $\rightarrow$ `a, b := b, a` $\gg$ $\|$ $\ll$ `b < c` $\rightarrow$ `b, c := c, b` $\gg$

is an expression that when repeatedly executed sorts `a`, `b`, and `c` into descending order. The asynchronous combinator, $\|$, is associative and commutative.

The **product combinator**, $*$, specifies that the two expressions must be performed as a single, atomic action. For example,

$\ll$ C_1 $\rightarrow$ `a := c` $\gg$ $*$ $\ll$ C_2 $\rightarrow$ `b := d` $\gg$

is equivalent to

$\ll$ $C_1 \wedge C_2$ $\rightarrow$ `a, b := c, d` $\gg$

If both preconditions are satisfied, then both actions can be performed as a single atomic action. Otherwise, no action will be taken. The product combinator, $*$, is associative and commutative. The product combinator distributes over the asynchronous combinator:

$$t_1 * (t_2 \| t_3) \;\equiv\; (t_1 * t_2) \| (t_1 * t_3)$$

`Synchronized Transitions` has a third combinator, the **synchronous combinator**, $+$. As this combinator is not used in any of the examples presented in this paper, it is not discussed further here.

2.2 A logic of `Synchronized Transitions` programs

A `Synchronized Transitions` program, P, is a collection of state variables, V_P and a transition valued expression, e_p. The transition valued expression e_p can be viewed as the root of a tree whose leaves correspond to transitions, and whose non-leave nodes correspond to expressions formed by one of the combinators. All transition valued expressions in P are nodes of this tree. Let E_P denote the nodes of this tree.

Executing a program, P, produces a sequence of states. A state transition is performed by selecting some subset of the transitions in the program and performing their multiassignments. The preconditions of the transitions and the combinators of e_P define which transitions can be

[1]In a hardware implementation, the levels (e.g. voltages) of many signals throughout the circuit may change simultaneously. Formal definition of mappings from signal levels to program states is outside the scope of this paper (see [GS88, SG88] for examples).

$$
\begin{aligned}
&\mathcal{P}(S_1, S_2, e, X) = \\
&\quad (\mathtt{e} \in X) \Rightarrow (C(S_1) \wedge (l(S_2) = r(S_1))) \\
&\qquad \text{if } e \;=\; \ll\ \mathtt{C} \rightarrow \mathtt{l} \ \mathtt{:=}\ \mathtt{r} \gg \\
&\quad ((\mathtt{e} \in X) = ((e_1 \in X) \vee (e_2 \in X))) \wedge \neg((e_1 \in X) \wedge (e_2 \in X)) \\
&\qquad \text{if } e \;=\; e_1 \| e_2 \\
&\quad (e \in X = e_1 \in X) \wedge (e \in X = e_2 \in X) \\
&\qquad \text{if } e \;=\; e_1 * e_2
\end{aligned}
$$

Figure 1: Predicates for transition valued expressions

performed together in a single state transition. The semantics of P can be defined by a relation $R_P(S_1, S_2)$ that holds if and only if state S_2 is a legal, immediate successor of S_1 in an execution of P.

Because the asynchronous combinator, $\|$, specifies non-deterministic selection of either operand for execution when both are enabled, a state may have more than one possible successor state. Thus, R_P is a relation, but not a function. In the following definition of R_P, this non-determinism is expressed by introducing an existentially quantified variable, X, that represents which transition valued expressions are executed in the transition from S_1 to S_2. A transition is include in this set if its multiassignment is performed; an expression formed with one of the combinators is included if either or both of its operands is executed.

The relation R_P is written as the conjunction of two clauses. The first clause states that the multiassignments of the executed transitions are performed, and that this set of executed transitions is allowed by the preconditions of the transitions and the combinators of e_P. This clause is written using a predicate, $\mathcal{P}$ which is defined in figure 1. If `x` is a variable or expression appearing in P, $x(S)$ denotes the value of `x` in state S. The first clause of R_P is

$$\exists X \subseteq E_P : \forall e \in E_P : \mathcal{P}(S_1, S_2, e, X)$$

The second clause of R_P states that variables that are not written by any of the executed transitions retain their old values. Let $W(v)$ denote the transitions that write state variable v. The second clause of R_P is

$$\forall v \in V_P : (W(v) \cap X = \emptyset) \Rightarrow (v(S_2) = v(S_1))$$

Combining this with the clause for transition valued expression yields R_P:

$$
\begin{aligned}
R_P(S_1, S_2) \;=\; \exists X \subseteq E_P : \quad & (\forall e \in E_P : \mathcal{P}(S_1, S_2, e, X)) \\
& \wedge (\forall v \in V_P : (W(v) \cap X = \emptyset) \Rightarrow (v(S_2) = v(S_1)))
\end{aligned}
$$

The definition of R_P lacks any sense of "fairness." In fact, the program is not required to do anything; $R_P(S, S)$ is satisfied for any S by choosing $X = \emptyset$. Therefore, liveness properties cannot be proven in this logic for `Synchronized Transitions`. The choice to focus on safety properties is discussed further in [SG90].

A more serious omission is the possibility that two transitions that write different values to the same state variable may be combined with the $*$ combinator. The definition of R_P above prevents such a pair of transitions from being selected for simultaneous execution. This definition is unsatisfying for two reasons. First, `Synchronized Transitions` is a hardware description language, and it is not clear how an electronic circuit should be implemented so as suppress conflicting assignments. Second, it is intended that `Synchronized Transitions` should be useful for simulation as well as formal verification. If conflicting assignments may be present but can't be executed, then deciding that a state has a legal successor is an NP-complete problem. For these reasons, it is preferable to forbid programs with conflicting assignments. It is straightforward to derive an "abort condition" by methods analogous to those used for R_P. This condition is called the "Exclusive-Write" condition in [SG90]; a similar condition is described, but not named, in [CM88].

2.3 Invariants and proof rules

Given the definition of R_P above, it is straightforward to define general properties of programs and rules for manipulating transition valued expressions and proving assertions about programs. Many of the formal verification results for designs represented in `Synchronized Transitions` are based on invariants [LS84]. A property, I, is an invariant of a program, P, if I holding in one state guarantees that I will hold in all future states. In particular, I is an invariant if and only if:

$$(I(S_1) \wedge R_P(S_1, S_2)) \Rightarrow I(S_2)$$

In general, an invariant could be established for any program by directly verifying the above formula. However, it is often simpler to prove properties of an entire class of programs and then use these properties to simplify the proof of a specific invariant.

Property 1 *Let $P = (V_P, e_P)$ such that $e_P = e_1 \| e_2$.*
Let $P_1 = (V_P, e_1)$, and $P_2 = (V_P, e_2)$.
If I is an invariant of P_1 and P_2, then I is an invariant of P.

Proof: Let S_1 and S_2 be two states such that $I(S_1) \wedge R_P(S_1, S_2)$.
By the definition of R_P,

$$\begin{aligned} \exists X \subseteq E_P : \quad & (\forall e \in E_P : \mathcal{P}(S_1, S_2, e, X)) \\ & \wedge (\forall v \in V_P : (W(v) \cap X = \emptyset) \Rightarrow (v(s_2) = v(s_1))) \end{aligned}$$

Consider the clause for e_P,

$$\mathcal{P}(S_1, S_2, e_P, X) = ((\mathsf{ep} \in X) = ((e_{P1} \in X) \vee (e_{P2} \in X))) \wedge \neg((e_1 \in X) \wedge (e_2 \in X))$$

It is straightforward to show by induction on the height of the expression tree for any program, that $(X \neq \emptyset) \equiv (e_P \in X)$. Therefore, if $e_P \notin X$, $S_2 = S_1$, and $I(S_2)$ trivially. Otherwise $e_P \in X$, which implies $(e_1 \in X) \vee (e_2 \in X)$. If $(e_1 \in X)$ then $(e_2 \notin X)$, and it follows that X is a subset of E_{P1}. Therefore, $R_{P1}(S_1, S_2)$, and $I(S_2)$ because I is an invariant of $P1$. Otherwise, $(e_2 \in X)$, and

$I(S_2)$ follows by an analogous argument. Therefore, I is an invariant of P.
□

A common application of property 1 is to establish invariants of programs that are composed using only the $\|$ combinator. For such programs, each transition can be considered as a separate program. Establishing the invariant for each transition, establishes it for the entire program.

Many other rules for manipulating `Synchronized Transitions` expressions can be derived using the semantics of transition valued expressions given in the previous section. For example, the associative, commutative, and distributive properties of the combinators mentioned previously follow directly from the definition of R_P. The timing verification presented in this paper uses the following property that was first mentioned in section **??**.

Property 2

$$\begin{array}{lrcl} \textit{Let} & t_1 & = & \ll \texttt{C1} \rightarrow \texttt{l1 := r1} \gg \\ & t_2 & = & \ll \texttt{C2} \rightarrow \texttt{l2 := r2} \gg \\ & t_3 & = & \ll \texttt{C1 AND C2} \rightarrow \texttt{l1, l2 := r1, r2} \gg \\ & e & = & t_1 * t_2 \end{array}$$

Let P be a program that contains e, and P' be the same program with e replaced by t_3.
$R_P = R_{P'}$.

Proof: It suffices to show $R_P(S_1, S_2) \Rightarrow R_{P'}(S_1, S_2)$, and $R_P(S_1, S_2) \Leftarrow R_{P'}(S_1, S_2)$. First, assume $R_P(S_1, S_2)$. By the definition of R_P,

$$\begin{array}{ll} \exists X \subseteq E_P : & (\forall e \in E_P : \mathcal{P}(S_1, S_2, e, X)) \\ & \wedge (\forall v \in V_P : (W(v) \cap X = \emptyset) \Rightarrow (v(s_2) = v(s_1))) \end{array}$$

If $e \notin X$, then $X \subset E_{P'}$, and it is straightforward to show $R_{P'}(S_1, S_2)$. If $e \in X$, let $X' = (X - \{t_1, t_2, e\}) \cup \{t_3\}$.

$$\begin{array}{rcl} \mathcal{P}(S_1, S_2, e, X) & \Rightarrow & (e \in X = t_1 \in X) \wedge (e \in X = t_2 \in X) \\ & \Rightarrow & (t_1 \in X) \wedge (t_2 \in X) \\ & \Rightarrow & C_1(S_1) \wedge C_2(S_2) \wedge (l_1(S_2) = r_1(S_1)) \wedge (l_2(S_1) = r_2(S_1)) \\ & \Rightarrow & \mathcal{P}'(S_1, S_2, t_3, X') \end{array}$$

Therefore,

$$(\forall e \in E_P : \mathcal{P}(S_1, S_2, e, X)) \Rightarrow (\forall e \in E_{P'} : \mathcal{P}'(S_1, S_2, e, X'))$$

Since anything written by t_1 or t_2 is written by t_3,

$$\forall v \in V_P : (W(v) \cap X = \emptyset) = (W'(v) \cap X' = \emptyset)$$

It follows that $R_{P'}(S_1, S_2)$ as required. Therefore, $R_P(S_1, S_2) \Rightarrow R_{P'}(S_1, S_2)$. The argument that $R_{P'}(S_1, S_2) \Rightarrow R_P(S_1, S_2)$ is analogous.
□

Program structuring

Programs that describe real designs can include thousands of transitions. `Synchronized Transitions` provides three mechanisms for structuring large programs: modules, cells, and quantified expressions. Modules are analogous to those of other modular languages (e.g. Modula-2 [Wir85]) and are not discussed further here.

Cells encapsulate parts of the design. A cell may contain local state variables, transitions, and other cells. The formal parameters of a cell can be either state variables of static expressions. State variable parameters are shared with the instantiating cell and provide communication between cells. The values of static expression parameters are bound when the cell is instantiated and can be used to control recursive instantiations or to determine the sizes of arrays. Syntactically, cell definitions resemble function definitions in other languages. Likewise, cell instantiations syntactically resemble function invocations. The value of an instantiation is a transition valued expression, the transitions of the cell. This expression can be composed with other expressions using any of the combinators.

The syntax for a quantified expression is {*op IndexDeclarations* `|` *expr*}. Each index must be of a bounded, discrete type (e.g. an integer subrange). The quantified expression is equivalent to evaluating *expr* for each value of each of the indices, and combining the results with *op*. Any associative and commutative operator may be used for *op*. For example, with the type definition

```
TYPE T = INTEGER SUBRANGE 1..8;
```

the quantified expression {`+ i: T | i`} has the value of 36, and the expression

```
{* i: T | << y(i) := y(i-1) >> }
```

describes an eight-bit, synchronous shift register.

3 The STARI chip

This section describes the communications chip which provided the examples presented in this paper. This chip is being built to demonstrate a novel circuit technique for interprocessor communication. This technique, called STARI [GL90] (Self-Timed At Receiver's Input) combines self-timed and synchronous circuit techniques. Figure 2 shows a STARI interface. The key component of STARI is a self-timed FIFO that receives data from a synchronous transmitter and delivers data to a synchronous receiver. The transmitter and the receiver operate at the same clock rate; however, these clocks are not required to be in the same phase. STARI eliminates the skew problems that limit the speed of synchronous designs. During each cycle (period of the global frequency reference), the transmitter sends one completion encoded datum [Sei79] that is inserted into the FIFO upon arrival. Likewise, the receiver removes one item from the FIFO each cycle. Once properly initialized, the FIFO never overflows or underflows.

Correct operation of the STARI circuit requires that the FIFO can complete each insert and remove operation within one cycle. When this requirement is met, the FIFO appears as a synchronous component to both the transmitter and the receiver, and both the transmitter and the receiver appear to the FIFO as well-behaved self-timed systems. The transmitter produces a new

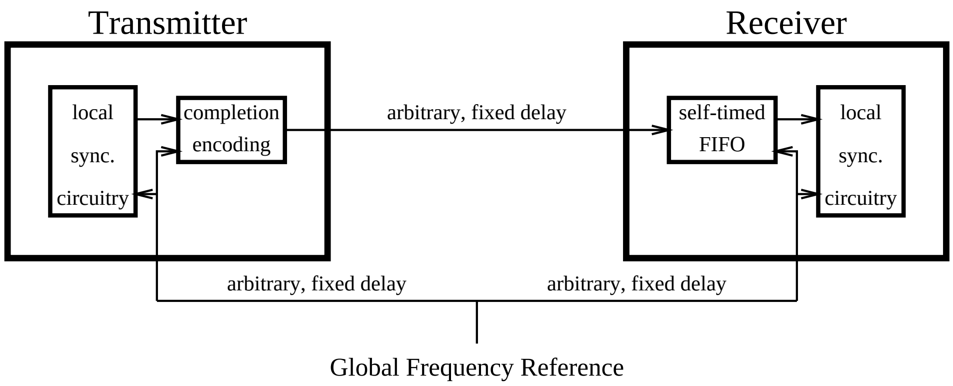

Figure 2: STARI communication

data value each clock cycle, just as if the FIFO were another component synchronous to its own clock. Because the FIFO performs each insert operation within one cycle, the FIFO issues a request for data before the corresponding datum arrives. Thus, the synchronous transmitter satisfies the self-timed signaling conventions of the self-timed FIFO. Likewise, the prompt response of the FIFO to requests from the receiver guarantees that the receiver does not issue a new request until the previous one has been satisfied. Therefore, the interface between the FIFO and the receiver is correctly timed.

FIFO's are often used to interface circuits operating with independent clocks. Typically, the FIFO generates flow control signals for the transmitter and receiver. These signals direct the transmitter to suspend sending if the FIFO is nearly full, and the receiver to stop requesting data if the FIFO is nearly empty. These signals must be synchronized to the clocks of the transmitter and receiver respectively and introduce the possibility of synchronization failure [CM73]. Instead of using flow control signals, static timing analysis guarantees that the FIFO in STARI can neither overflow nor underflow. This allows STARI to operate at without the possibility of synchronization failures and at higher rates than traditional FIFO based designs.

The STARI chip (see figure 3) functions as a multiplexer/demultiplexer, interfacing the local memory of a processor node to high-speed coaxial cable links. Inside the chip, a STARI receiver acquires data from the ring; a one-to-ten demultiplexer converts this data into 10 bit words[2]. Each word consists of 8 bits of data for the memory, one bit of control information, and one bit of parity. Data can be written into the local memory, read from the local memory, or forwarded to the next node of the ring. Control information is pipelined through an off-chip ring controller, and an internal shift register delays the data to match the controller's latency. The multiplexer and STARI transmitter send data from the local memory into the high speed ring.

Other communication schemes that depend on close or exact frequency matching at the transmitter and receiver but are tolerant of phase variations have been implemented [HN86, JH88,

[2]In the actual chip, the demultiplexer precedes the STARI receiver for performance reasons (see section 5).

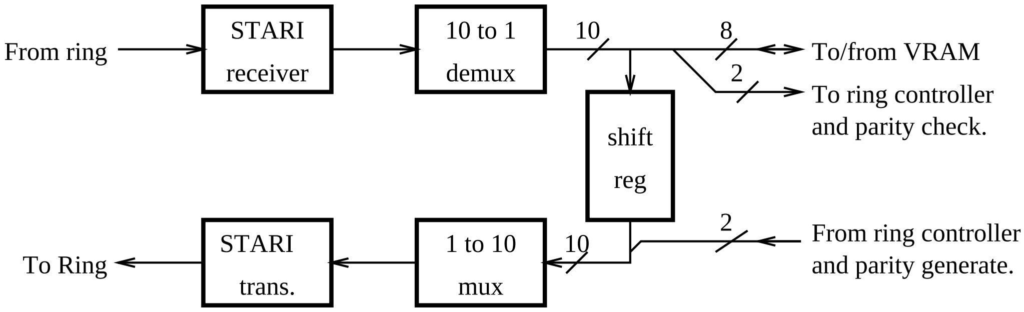

Figure 3: The STARI chip

AMD89]; these schemes use clock forwarding or employ a phase-locked loop to regenerate the transmitter's clock at the receiver. A thorough comparison of STARI with these schemes is outside of the scope of this paper. The author of this paper is not aware of any formal verification for other approaches[3].

The arguments in this section for the correctness of STARI are informal; a rigorous proof requires formal descriptions of the transmitter, the receiver, the FIFO, and their timing requirements. In this paper, `Synchronized Transitions` is used to express formal descriptions of these components, and the timing correctness of STARI is verified by proving a suitable invariant. The next section describes this verification.

4 Verifying STARI

The novelty of STARI is that it provides a communication path between two circuits operating at the same clock frequency, it allows the transmitter's and receiver's clock to be skewed, and it is free from synchronization failure. This section describes the verification of these properties. To verify STARI, a communication link including the transmitter, the FIFO, and the receiver must be described. This description must include both the logical behavior of these components and their timing behavior. Next a specification of correct behavior must be given. The verification task is to show that the description of the implementation satisfies the specification. Finally, as verification is with respect to a particular model and a particular specification, the consequences of these abstractions must be considered to identify some of the aspects of the design that *are not* covered by the formal verification.

[3]Gordon has published a correctness proof for the network interface chip used in the Cambridge Fast Ring [GH85]. Gordon's proof assumes that bits from the ring behave like other synchronous signals and does not consider the off-chip phase locked loop used for clock extraction(see [HN86]).

Describing STARI

Appendix A is a `Synchronized Transitions` program for a STARI interface. The derivation of this program is presented here. The description of the logical behavior is presented first; then, the parts of the program associated with time are presented.

STARI is based on transferring data according to a self-timed protocol. In such a protocol, it must be possible to distinguish successive data values. In the current implementation, this is done with a dual-rail-code where each logical value, `q`, is encoded with two signals, `q.t` and `q.f` as shown below:

q.t	q.f	q
low	low	empty
low	high	FALSE
high	low	TRUE
high	high	illegal

As a shorthand, we write

$$\mathtt{valid(q)} \equiv ((\mathtt{q} = \mathtt{FALSE}) \vee (\mathtt{q} = \mathtt{TRUE}))$$

In operation, a dual-rail-code signal alternates between `empty` and `valid` values. The `empty` values function as spacers between `valid` values allowing a circuit, for example, to distinguish between two consecutive `TRUE` values and one value that is asserted for a long time.

In the program of Appendix A, data from the transmitter, in the FIFO, and the most recent value acquired by the receiver are represented by `y`, an array of dual-rail-code signals. In particular, `y(0)` is the value currently output by the transmitter, `y(1)` through `y(n)` are the values held by the `n` FIFO stages, and `y(n+1)` is the value most recently acquired by the receiver.

The logical behavior of the transmitter is to assign values to `y(0)` such that these assignments alternate between `valid` and `empty` values. This is accomplished by the transition

```
≪ valid(nexty0) ≠ valid(y(0)) → y(0) := nexty0 ≫
```

where `nexty0` is a variable that can take on any legal dual-rail-code value as described by the following transitions:

```
   ≪ nexty0.t, nexty0.f := FALSE, FALSE ≫  (* empty *)
|| ≪ nexty0.t, nexty0.f := TRUE,  FALSE ≫  (* Boolean true *)
|| ≪ nexty0.t, nexty0.f := FALSE, TRUE ≫   (* Boolean false *)
```

The FIFO operates according to the following rule: a stage may acquire its predecessors value when its successor has acquired its current value. This is expressed by the transition:

```
≪ (valid(y(j-1)) ≠ valid(y(j))) AND (valid(y(j)) ≠ valid(y(j+1)))
      → y(j) := y(j-1)
≫
```

For a more thorough discussion of the FIFO implementation, including a proof that it corresponds to a delay-insensitive circuit, see [SG90].

The receiver acquires values from the FIFO as described by the following transition:

```
≪ y(n+1) := y(n) ≫
```

To describe real-time properties of the implementation, time is introduced using auxiliary variables [OG76]. These variables are declared in the "`AUXILIARY`" part of the program to indicate that they do not correspond to signals of the physical circuit. For the purposes of verification, auxiliary variables can be regarded as additional state variables. The current time is represented by the real valued variable τ. Furthermore, an array of records, `t`, is used for time related bookkeeping: `t(j).a` is the time at which `y(j)` was assigned its current value; `t(j).i` is the time at which this value was assigned to `y(0)`.

In general, there are two kinds of timing properties:

Timing lower bounds: a transition is not performed until after a specified time.

Timing upper bounds: a transition is guaranteed to be performed by a specified time.

Timing lower bounds can be expressed by strenthening the transition's precondition. For example,

```
  ≪ ... → y(j) := y(j-1) ≫
* ≪ τ ≥ max(t(j-1).a, t(j+1).a) + mind ≫
```

states that `y(j)` can only be updated if both `y(j-1)` and `y(j+1)` received their current values at least `mind` (the minimum delay) time units ago.

Timing upper bounds are somewhat more subtle because `Synchronized Transitions` provides no mechanism for stating liveness properties of a transition (e.g. conditions under which it *must* be performed). Instead, time is considered to be part of the environment; in other words, the program specifies properties of time but has no pretensions of implementing it. As noted in [Lam86], real time properties can be stated as safety properties; this approach is taken here. In particular, assertions are added to the program to state that the current time can not exceed a certain value until after some enabled transition is performed. This is equivalent to stating that once enabled, the transition is performed within the specified time limit. These assertions are written by adding an "`ENVIRONMENT`" part to the program. The `ENVIRONMENT` part is an assertion about properties that the program's environment is assumed to maintain. In this paper, the `ENVIRONMENT` part is used to state properties of time. For example,

```
ENVIRONMENT
  (     (valid(y(j-1)) ≠ valid(y(j)))
    AND (valid(y(j))   = valid(y(j+1)))
  ) IMPLIES (τ ≤ max(t(j-1).a, t(j+1).a) + maxd)
```

states that the transition that writes `y(j)` will be performed within `maxd` time units after becoming enabled.

Verifying the program

The key to the verification is to show that the FIFO accepts each input before the next one arrives, and that the FIFO delivers a new output before the next request arrives. To state this more formally, let

$$\begin{array}{rrcl} & Q1 & \equiv & (\tau = t(0).a + p) \;\Rightarrow\; (y(1) = y(0)) \\ \text{and} & Q2 & \equiv & (\tau = t(n+1).a + p) \;\Rightarrow\; (\mathrm{valid}(y(n+1)) \neq \mathrm{valid}(y(n))) \end{array}$$

$Q1$ states that the FIFO accepts each datum from the transmitter before the next one arrives. $Q2$ states that the FIFO satisfies each request for data from the receiver before the next request is issued. Verifying these two properties establishes the timing correctness of the program. Appendix B presents the details of this verification; an overview of the approach is presented here. The timing verification is based on an invariant, and it is shown that this invariant implies $Q1$ and $Q2$.

To characterize of the STARI implementation, the following parameters are used:

p = period of the global frequency reference, $0 < p$
d = `maxd`, maximum delay of a FIFO stage, $0 < d < p/2$
n = number of FIFO stages, $0 < n$
x = latency, time difference between arrival of a datum from the transmitter and removal of the same datum by the receiver.
$nd + (p - d) \leq x \leq (n+1)(p-d)$

The bounds on x guarantee that data is available on the FIFO output, `y(n)`, $p - d$ time units before it is acquired by the receiver's input latch, `y(n+1)`.

The invariant of Appendix B is the conjunction of seven clauses, labeled $I.1$ through $I.7$. $I.1$ asserts that for `y(0)` the insertion time, `t(0).i`, and assignment time,`t(0).a` are the same, which corresponds to the intuitive definition of the array `t`. $I.4$ states that if two adjacent elements of the `y` array are both valid, then they have identical values. $I.7$ asserts that `t(n+1).i`, the insertion time for `y(n+1)`, is exactly x less than the assignment time `t(n+1).a`, which corresponds to the intuitive definition of x (the FIFO latency).

Clauses $I.2$ and $I.3$ give a schedule by which values must arrive at each stage of the FIFO. This schedule is defined by a function σ. $I.2$ asserts

$$t(k).a \leq t(k).i + \sigma(k)$$

Intuitively, $I.2$ says that if a value is assigned to `y(k)`, this assignment must occur within $\sigma(k)$ time units of the time at which it was originally assigned to `y(0)`. $I.3$ asserts

$$\tau \leq t(k).i + \sigma(k) + p$$

Intuitively, $I.3$ says that the value held by a stage can't be too old (i.e. have a too small value for $t(k).i$). Clauses $I.5$ and $I.6$ of the invariant assert that two adjacent values in the array `y` must either correspond to the same value (same insertion times) or to consecutive values of the input sequence (insertion times differ by p).

Like most proofs of invariants, the proof presented in Appendix B is mostly a matter of using consequences of the invariant in the precondition of a transition to show that the invariant continues to hold after the transition executes. The proof has two distinctive features: first, the consequences of the invariant in the precondition include systems of inequalities, not just Boolean relationships; second, in addition to considering each transition, it must also be shown that any legal change of the global time variable, τ, maintains the invariant. As described above, changes to τ are not modeled by transitions, instead, constraints on the legal values of τ are given in the "`ENVIRONMENT`" section of the program.

Consequences of the verification

The proof in Appendix B provides explicit timing bounds under which STARI communication is reliable. These bounds illuminate how the STARI link operates.

For most applications, x, the exact latency of the communication link is not critical; the link will continue to operate if x is increased or decreased by an integral multiple of p. However, it is important that the link can tolerate differences in the phase of the global frequency reference between the transmitter and the receiver. This difference is analogous to the skew of synchronous systems and is denoted by s in the following. By ignoring integral multiples of p, s can be assumed to be between 0 and p. Correct operation is possible if there is a value for x such that

$$\begin{array}{rl} & nd + (p - d) \leq x \leq (n + 1)(p - d) \\ \text{and} & x \bmod p = s \end{array}$$

Such a value is guaranteed to exist if

$$(n + 1)(p - d) - (nd + (p - d)) \geq p$$

which is equivalent to

$$n(p - 2d) \geq p$$

The timing model presented here is very simplistic. There are three main criticisms of the model. First, the synchronous operations of the transmitter and of the receiver are assumed to occur at times separated by exactly p. Second, the delay model for the FIFO stage is simplistic. Finally, although time is modeled as a real number, operations are assumed to be atomic; the underlying physical (i.e. analog) circuitry is ignored. Some of these limitation could be overcome with a more detailed model. For example, clock jitter and drifting skew could be added to the models of the transmitter and receiver. The analysis would be tedious, but similar methods could be used. In practice, many of these effects can be overcome by increasing the number of stages in the FIFO to increase tolerance of timing uncertainties. Other problems, such as neglecting analog aspects of the circuitry are outside of the simple modeling methods presented in this paper.

Being based on invariants, the proof is one of a safety property. Even though the proof demonstrates that certain predicates are satisfied before τ, the time variable, reaches certain values, this does not establish the liveness of STARI. This is because there is no formal proof that the timing constraints allow τ to increase without bound. This is a modeling issue: any realistic model of the chips components will ensure that time can continue to increase because real circuits do not

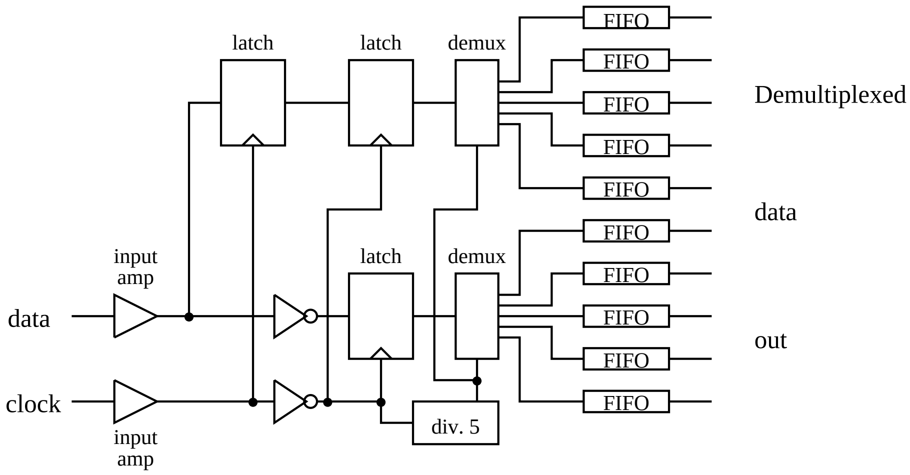

Figure 4: Revised input for STARI chip

arrest the forward flow of time. However, such properties cannot be proven without including some form of fairness to the semantics of `Synchronized Transitions`. Furthermore, the proof does not consider initialization, i.e. how the invariant is first established. Because initialization involves progress from an unknown state to one that satisfies the invariants, initialization is a liveness property and beyond the scope of the proof methods presented in this paper.

Finally, it is interesting to note what doesn't appear in the proof. The proof makes no mention of `mind`, the minimum propagation delay for a FIFO stage; that is, a stage can't be too fast. Thus, the timing specification for the FIFO stage only needs to include maximum delay times; such circuits are much easier to design than ones where both maximum and minimum delays are specified. Related to the absence of proving liveness properties is the fact that the timing conditions do not constrain time to be monotonically increasing. In this model, if time goes backwards, some enabled transitions may become disabled, but it will not enable any disabled ones. Since only safety properties are verified here, such anomalies are innocuous.

5 Simulating STARI

STARI allows two processors to communicate with neither the problems of clock skew from which synchronous systems nor the overhead of handshaking each data transfer which limits the performance of self-timed communication schemes. In fact, more than one data value may be in transit between the final latch of the transmitter and the first stage of the FIFO. This is a form of analog pipelining in the buffers and transmission lines connecting the transmitter to the receiver. This allows higher transfer rates than can be sustained by a self-timed FIFO. To maximize the performance of the STARI link, the data stream from the transmitter is demultiplexed into several slower streams before being input to the FIFO (see figure 4).

Numerous analog circuit techniques are employed in the design of the STARI chip including low-voltage, controlled impedance signaling [KK88] and analog feedback in the input amplifier. The chip is designed in 2μ CMOS to operate with a 350 MHz global frequency reference. Two data transfers are performed during each period of the frequency reference; thus, a transfer rate of 700 Mbits/sec is achieved for each data line.

Currently, formal techniques for verifying high-speed analog circuits such as those used in the STARI chip do not exist. Detailed circuit simulation is typically used for such circuits. CAzM [EK$^+$89] has been used in the design of the STARI chip. `Synchronized Transitions` is still valuable as a design language. Although these circuits are intrinsically analog, the behavior of each subcircuit has a simple discrete interpretation. Thus, the initial design can be described using `Synchronized Transitions`, and the transistor level implementation can then be simulated with a circuit simulator such as CAzM.

Using both `Synchronized Transitions` and a circuit simulator has several advantages. First, most circuit simulators produce waveforms as their output. Although waveforms are very useful for measuring fractions of nanoseconds and volts, they often lack intuitive significance in terms of the desired high level behavior. As an example, consider a pair of communicating STARI chips. The transmitter's multiplexer and the receiver's demultiplexer must be aligned in their operations; otherwise, the bits of each word that is transmitted will be shuffled with bits of the preceding and following words. Using `Synchronized Transitions`, auxiliary variables can be used to tag each bit with its position in the input stream. With these tags, it is easy to determine that each stage of the chip functions correctly. Then, each waveform of the circuit simulation can be compared with its discrete counterpart from the `Synchronized Transitions` simulation. This approach was very helpful at several steps of the STARI design, especially for addressing issues of initializing the chip.

Conversely, the circuit simulation provides validation[4] of the models used when writing the `Synchronized Transitions` program. For example, circuit simulation revealed a combinational path from the chip's reset signal through the divide-by-five counter and a transparent latch in the demultiplexer that was not correctly modeled in the original `Synchronized Transitions` program. The `Synchronized Transitions` program, like most circuit descriptions used by formal methods, is at the gate and latch level. Errors in these models invalidate the results of formal verification.

Simulation and Verification

One of the most attractive features of `Synchronized Transitions` is that the same description can be used for both simulation and verification. As this paper has been written for a workshop on formal methods, it is natural to ask: "Why simulate?"

Simulation and formal verification are not mutually exclusive approaches to validating a design. The objective of most hardware design is to maximize the probability that the final circuit will implement the desired function and to complete the design according to some project schedule. Informally, the specification includes a deadline for completing the design. Both simulation and formal verification can increase the probability of achieving a correct design within the available time. Reasons to simulate include:

[4] In this paper, the term "validation" refers to informal methods of verification.

- Formal techniques may not exist.
  Existing techniques for formal verification are not applicable for some circuits, for example, the high speed analog circuits of the STARI chip.

- Simulation can be fast and adequate.
  If sufficient confidence in the correctness of some part of the design can be achieved by simulation in less time than constructing a formal proof, more time is available to verify or simulate other parts of the design. For example, simulating STARI's demultiplexer quickly confirms that bits are demultiplexed to the correct positions. Using auxiliary variables to tag these bits raises the level of confidence even higher. Extending this approach even further, Bryant has recently described how simulation can be used as a part of formal verification [Bry91].

- The formal specification may be incomplete.
  For example, no liveness properties were verified for STARI. Other important issues that are beyond the scope of current formal techniques include power consumption, noise immunity, and manufacturing yield.

- The formal specification may not exist.
  Most engineers reason about circuits informally and operationally and are not fluent in the assertional approach of formal methods. Simulation provides a common language for communicating with traditional engineers.

- Simulation can find counterexamples quickly.
  After deriving a circuit from its specification, simulation can quickly uncover errors in the derivation. This is analogous to "plugging numbers into a formula" after an algebraic derivation. Such simulation allows the designer to avoid spending time attempting to verify erroneous designs.

6 Conclusions

This paper described how `Synchronized Transitions` has been used for simulation and formal verification of a communication chip. The principle contribution of this paper is a correctness proof for the timing of STARI. Furthermore, rules for translating `Synchronized Transitions` programs to logic expressions have been given, and applications of `Synchronized Transitions` as a simulation language have been presented.

The formal arguments presented in this paper were derived without the assistance or checking of a mechanical theorem prover. Many people, including the author, consider manual proofs to be less reliable than those verified by computer proof systems. Inspite of these limitations, manual proofs can be used to illustrate methods of reasoning about hardware, including methods that are not available in current automated systems. For example, the arguments in the proof for STARI included reasoning about linear inequalities of real-valued variables; most theorem proving systems for hardware verification do not include a theory of the real numbers. Such a theory could be added to most of these systems, but a complete development of this rich body of mathematics would be a daunting undertaking for a user of any of the existing theorem proving programs.

On the other hand, the proof in this paper required only a few properties of inequalities and arithmetic. Examples such as this one can suggest practical extensions to existing methods of automated theorem proving.

`Synchronized Transitions` programs have no hidden state (e.g. program counters). The set of legal successor states is completely determined by the current state. This transparency of programs admits a simple transformation from transition valued expressions to logic predicates. The transformations presented here for transitions and expressions formed with the asynchronous combinator, $\|$, are similar to those presented in [SGG89]. This presentation extends this approach to expressions formed using arbitrary combinations of the asynchronous and product combinators. In [Gre91], transformations for expressions using any combination of all three combinators, including the synchronous combinator, $+$, were presented. Deriving proof rules for classes of programs other than those discussed in this paper is an area for further research. The transformation from a program to its predicate is efficient; the size of the logic formula is linear in the size of the `Synchronized Transitions` expression. Thus, these transformations yield a formula that is practical for use by theorem proving programs (e.g. [Gor85, BM88, GG89, BDV90]).

`Synchronized Transitions` is also useful as a simulation language, This paper sketched the application of `Synchronized Transitions` to the simulation of the chip. Simulation and formal verification are complementary methods of achieving correct hardware designs. Formal methods are particularly appropriate for timing verification because the continuous nature of time makes exhaustive simulation impossible. Simulation, on the other hand, is essential in real designs to validate specifications, and to test circuits and properties that are beyond the scope of existing formal methods.

Acknowledgements

The author wishes to express his appreciation to the many people whose assistance has been valuable in this work. `Synchronized Transitions` is the creation of Jørgen Staunstrup, and I have benefited from continuing interaction with Jørgen. Leslie Lamport explained to me how to specify time with constraints and real-time properties as invariants. Chuck Thacker, Jim Sturm, and Steve Nugent made suggestions that have resulted in a much higher performance design for the STARI chip, and thereby raised many questions about verification in the presence of analog circuits. Andrew Appel assisted in formulating the semantics for `Synchronized Transitions` presented in section 2. Thanks to the referees for many useful comments on an earlier draft of this paper. Financial support for this work has been through an AT&T predoctoral fellowship and summer internships at DEC Systems Research Center and Matsushita Information Technology Laboratory.

References

[AMD89] AMD Inc. *The SUPERNET Family for FDDI*, 1989.

[BDV90] David A. Basin and Peter Del Vecchio. Verification of combinational logic in Nuprl. In M. Leeser and G. Brown, editors, *Hardware Specification, Verification and Synthesis: Mathematical Aspects*, pages 333–357. Springer-Verlag, 1990. Lecture Notes in Computer Science, 408.

[BM88] R.S. Boyer and J.S. Moore. *A Computational Logic Handbook*. Academic Press, Boston, 1988.

[Bry91] Randal E. Bryant. A methodology for hardware verification based on logic simulation. *Journal of the ACM*, 38(2):299–328, April 1991.

[CM73] T.J. Chaney and C.E. Molnar. Anomalous behavior of synchronizer and arbiter circuits. *IEEE Transactions on Computers*, C-22(4):421–422, April 1973.

[CM88] K.M. Chandy and J. Misra. *Parallel Program Design: A Foundation*. Addison-Wesley, 1988.

[EK+89] D.J. Erdman, S.W. Kenkel, et al. CAzM: A numerically robust, table-based circuit simulator. Technical Report TR89-23, Microelectronics Center of North Carolina, Research Triangle Park, North Carolina, July 1989.

[GG89] Stephen J. Garland and John V. Guttag. An overview of LP: the Larch Prover. In *Proceedings of the Third International Conference on Rewriting Techniques and Applications*. Springer-Verlag, 1989. LNCS 385.

[GH85] Michael J.C. Gordon and John Herbert. A formal hardware verification methodology and its application to a network interface chip. Technical Report 66, Computing Laboratory, Cambridge University, 1985.

[GL90] Mark R. Greenstreet and Kai Li. Simple hardware for fast interprocessor communication. Technical Report CS-TR-242-90, Department of Computer Science, Princeton University, Princeton, NJ, January 1990.

[Gor85] Michael J.C. Gordon. HOL: a machine oriented formulation of higher order logic. Technical Report 68, Computing Laboratory, Cambridge University, 1985.

[Gre91] Mark R. Greenstreet. Using synchronized transitions for simulation and timing verification. Technical Report MITL-TR-01-91, Matsushita Information Technology Laboratory, Princeton, NJ, August 1991.

[GS88] Mark R. Greenstreet and Jørgen Staunstrup. Atomicity, programs, and hardware. Technical Report ID-TR: 1988-46, Department of Computer Science, Technical University of Denmark, Lyngby, Denmark, November 1988.

[GS90] Mark R. Greenstreet and Ken Steiglitz. Bubbles can make self-timed pipelines fast. *Journal of VLSI and Signal Processing*, 2(3):139–148, November 1990.

[HN86] Andy Hopper and Roger M. Needham. The Cambridge Fast Ring networking system. Technical Report 90, Computing Laboratory, Cambridge University, 1986.

[JH88] Mark G. Johnson and Edwin L. Hudson. A variable delay line PLL for CPU-coprocessor synchronization. *IEEE Journal of Solid-State Circuits*, 23(5):1218–1223, October 1988.

[KK88] Thomas F. Knight and Alexander Krymm. A self-terminating low-voltage swing CMOS output driver. *IEEE Journal of Solid-State Circuits*, 23(2):457–464, April 1988.

[Lam86] Leslie Lamport. A simple approach to specifying concurrent systems. Technical Report 15, Digital Equipment Corporation, Systems Research Center, Palo Alto, CA, December 1986.

[LS84] Leslie Lamport and Fred B. Schneider. The 'Hoare logic' of CSP, and all that. *ACM Transactions on Programming Languages*, 6(2), April 1984.

[OG76] Susan Owicki and David Gries. An axiomatic proof technique for parallel programs I. *Acta Informatica*, 6(1), 1976.

[RS87] Anders P. Ravn and Jørgen Staunstrup. Synchronized Transitions. Technical Report DAIMI PM-219, Computer Science Department, Aarhus University, Århus, Denmark, January 1987.

[Sei79] Charles L. Seitz. System timing. In *Introduction to VLSI Systems*, chapter (Carver Mead and Lynn Conway) 7, pages 218–262. Addison Wesley, 1979.

[SG88] Jørgen Staunstrup and Mark R. Greenstreet. From high-level descriptions to VLSI circuits. *BIT*, 28(3):620–638, 1988.

[SG90] Jørgen Staunstrup and Mark R. Greenstreet. Synchronized Transitions. In Jørgen Staunstrup, editor, *Formal Methods for VLSI Design*, chapter 2, pages 71–128. North-Holland, 1990.

[SGG89] Jørgen Staunstrup, S. Garland, and J. Guttag. Localized verification of circuit descriptions. In *Proceedings of the Workshop on Automatic Verification Methods for Finite State Systems, LNCS 407*. Springer Verlag, 1989.

[Wir85] Niklaus Wirth. *Programming in Modula-2*. Springer-Verlang, third edition, 1985.

Appendix A: Synchronized Transition program for STARI

```
STATIC  (* constants that parameterize the implementation *)
  n: INTEGER;  (* the number of FIFO stages *)
TYPE
  DualRailCode = RECORD
                   t, (* true means this is a Boolean TRUE *)
                   f: (* true means this is a Boolean FALSE *)
                     BOOLEAN
                 END;

  TimeRec = RECORD
              a, (* time of last assignment *)
              i: (* time this item was inserted into the FIFO *)
                REAL
            END;

  (* miscellaneous types *)
  StateIndex = INTEGER SUBRANGE 0..n+1;
  StageIndex = INTEGER SUBRANGE 1..n;
  DA = ARRAY(j: StateIndex): DualRailCode;
  TA = ARRAY(j: StateIndex): TimeRec;
  V = FUNCTION(x: DualRailCode): BOOLEAN;

STATIC  (* a utility function *)
  valid: V = BEGIN (x.t OR x.f) END;

STATE   (* variables modified as the program executes *)
  y: DA;   (* data values *)
  nexty0:  (* the next value for the transmitter to assign to y(0) *)
    DualRailCode;

AUXILIARY  (* bookkeeping for formal verification *)
  p,     (* period of the global frequency reference *)
  maxd,  (* maximum delay of a FIFO stage *)
  mind,  (* minimum delay of a FIFO stage *)
  tau:   (* the current time (an auxiliary) *)
    REAL;
  t: TA; (* bookkeeping of transition times *)
```

```
  ENVIRONMENT  (* predicates describing time *)
    (* these conditions guarantee that each operation completes
       within it's maximum delay time *)
        (tau <= t(0).a + p)    (* applying inputs *)
    AND { AND j: StageIndex |  (* bounds on delays in the FIFO *)
          (     (valid(y(j-1)) <> valid(y(j)))
            AND (valid(y(j))   =  valid(y(j+1)))
          )
          IMPLIES (tau <= max(t(j-1).a, t(j+1).a) + maxd)
        }
    AND (tau <= t(n+1).a + p)  (* consuming outputs *)

BEGIN  (* the transitions *)
  (* input to the FIFO *)
        (* y(0) alternates between valid and NOT valid *)
        << valid(nexty0)  <> valid(y(0)) -> y(0) := nexty0 >>
        (* a new value for y(0) exactly once each period *)
      * << tau = t(0).a + p -> t(0).a, t(0).i := tau, tau >>

  ||  (* generate new dual-rail-code values for nexty0 *)
          << nexty0.t, nexty0.f := FALSE, FALSE >> (* reset *)
      ||  << nexty0.t, nexty0.f := TRUE,  FALSE >> (* Boolean true *)
      ||  << nexty0.t, nexty0.f := FALSE, TRUE >>  (* Boolean false *)

  ||  (* the FIFO *)
      {|| j: StageIndex |
          (* logical behavior *)
          << (valid(y(j-1)) <> valid(y(j))) AND (valid(y(j)) = valid(y(j+1)))
              -> y(j) := y(j-1)
          >>

          (* minimum delays for the FIFO stages *)
        * << tau >= max(t(j-1).a, t(j+1).a) + mind -> t(j).a := tau >>
          (* maximum delays for the FIFO stages *)

          (* maintaining t(j).i *)
        * << t(j).i := t(j-1).i >>
      }

      (* consuming the FIFO's output, once each period *)
  ||    << tau = t(n+1).a + p -> t(n+1).a, t(n+1).i := tau, tau>>
      * << y(n+1) := y(n) >>
END;
```

Appendix B: Timing Verification of STARI

Appendix A presents a `Synchronized Transitions` program for STARI. In this appendix, an invariant that demonstrates the timing correctness of this program is stated and proved. The goal of this verification is to establish:

$$\begin{array}{rrcl} & Q1 & \equiv & (\tau = t(0).a + p) \Rightarrow (y(1) = y(0)) \\ \text{and} & Q2 & \equiv & (\tau = t(n+1).a + p) \Rightarrow (\text{valid}(y(n+1)) \neq \text{valid}(y(n))) \end{array}$$

$Q1$ states that the FIFO accepts each datum from the transmitter before the next one arrives. $Q2$ states that the FIFO satisfies each request for data from the receiver before the next request is issued. These two properties show that the FIFO satisfies the timing requirements of both the transmitter and the receiver, or from another perspective, they show that the behavior of the transmitter and the receiver comply with the self-timed protocol of the FIFO.

The following parameters characterize the FIFO, the transmitter, and the receiver.

p = period of the global frequency reference, $0 < p$
d = `maxd`, maximum delay of a FIFO stage, $0 < d < p/2$
n = number of FIFO stages, $0 < n$
x = latency, time difference between arrival of a datum from the transmitter and removal of the same datum by the receiver.
$nd + (p - d) \leq x \leq (n+1)(p-d)$

The bounds on x are derived by the following considerations. The minimum time for a value to propagate from `y(0)`, the input of the FIFO, to `y(n)`, the output of the FIFO, is nd. For the receiver to reliably acquire `y(n)`, x must be greater than nd plus the set-up time of the receiver's input latch. The greatest set-up time that STARI can guarantee is $p-d$, and this is the value used in the bound stated above. By making the provision for set up time less conservative, the lower bound for x could be reduced, with a bound of $nd \leq x$ for a physically unrealizable latch with no set-up requirements. The upper bound for x arises from the time needed for requests (corresponding to the $\text{valid}(y(j)) = \text{valid}(y(j+1))$ conjunct of the FIFO transition's precondition) to propagate from the output of the FIFO back to the input (see [GS90] for a more detailed description). In the following, it is shown these conditions are sufficient to guarantee correct operation.

The invariant

The timing verification of STARI presented here utilizes upper bounds on the times at which values arrive at FIFO stages. The intuition for these bounds comes from considering a FIFO where the delay of each stage is exactly d. In such a FIFO, the interarrival times for each stage are fixed at exactly p. The time between when a value is stored by one stage and when it is stored by the next is completely determined by d, x, and n. In particular, for some m ($-1 \leq m \leq n$) and some α ($d \leq \alpha < p - d$), the delay between stages is d for the first m stages (if $m > 0$), α for stage $m + 1$ (if $0 < m$ and $m < n$), and $p - d$ for the last $n - 1 - m$ stages (if $m < n$). To satisfy

$x = md + \alpha + (n-m)(p-d)$, let

$$\begin{array}{rcl} m & \equiv & \begin{cases} n\,, & x = nd + (p-d) \\ n - 1 - \left\lfloor \frac{x-(p-d)-nd}{p-2d} \right\rfloor\,, & x > nd + (p-d) \end{cases} \\ \alpha & = & x - md - (n-m)(p-d) \end{array}$$

These bounds for the fixed-delay scenario are not proven separately here. The FIFO with fixed delays is a special case of the more general FIFO with indeterminate delays; therefore, verifying the general FIFO verifies the fixed-delay version as a special case. Instead, the preceding description of the fixed-delay FIFO provides bounds for the behavior of the more general case. These bounds are used to formulate the invariant.

Let

$$\begin{array}{rcl} \delta(j) & \equiv & \begin{cases} 0\,, & 0 = j \\ d\,, & 1 \le j \le m \\ \alpha\,, & m+1 = j \\ p-d\,, & m+1 < j \le n+1 \end{cases} \\ \sigma(j) & \equiv & \displaystyle\sum_{k=0}^{j} \delta(j) \end{array}$$

$\delta(j)$ is the time between when $y(j-1)$ acquires a value and when $y(j)$ acquires it under the fixed-delay scenario, and $\sigma(j)$ is the accumulation of $\delta(j)$ from the FIFO's input. By the definitions of δ and σ, $\sigma(n+1) = x$.

In the following, T denotes the conditions maintained by the environment

$$\begin{array}{rcl} T & = & (\tau \le \mathtt{t(0).a} + \mathtt{p}) \\ & \wedge & \Big(\displaystyle\bigwedge_{1 \le j \le n} (\mathtt{valid(y(j\text{-}1))} \ne \mathtt{valid(y(j))}) \wedge (\mathtt{valid(y(j))} = \mathtt{valid(y(j\text{+}1))}) \\ & & \qquad \Rightarrow (\tau \le \max(\mathtt{t(j\text{-}1).a,\ t(j\text{+}1).a}) + d) \Big) \\ & \wedge & (\tau \le \mathtt{t(n\text{+}1).a\ +\ p}) \end{array}$$

Let

$$\begin{array}{rcll} I & \equiv & (t(0).i = t(0).a) & (I.1) \\ & \wedge & (\forall k \in \{0 \ldots n+1\} : & \\ & & \quad (t(k).a \le t(k).i + \sigma(k)) & (I.2) \\ & & \quad \wedge\ (\tau \le t(k).i + \sigma(k) + p) & (I.3) \\ & &) & \\ & \wedge & (\forall k \in \{0 \ldots n\} : & \\ & & \quad ((\mathrm{valid}(y(k)) = \mathrm{valid}(y(k+1))) & \\ & & \qquad\qquad \Leftrightarrow\ (y(k) = y(k+1))) & (I.4) \\ & & \quad \wedge\ ((y(k) = y(k+1)) \ \Rightarrow\ (t(k).i = t(k+1).i)) & (I.5) \\ & & \quad \wedge\ ((y(k) \ne y(k+1)) \ \Rightarrow\ (t(k).i = t(k+1).i) + p) & (I.6) \\ & &) & \\ & \wedge & (t(n+1).a = t(n+1).i + \sigma(n+1)) & (I.7) \end{array}$$

I is an invariant.

Proof: First it is shown that the program maintains I and T, and then it is shown that all actions of the environment, i.e. changes to τ, maintain I. To show that the program maintains I and T, property 1 in section 2.2 is used: each operand of the asynchronous combinator, $\|$, is considered independently. Property 2 is used to treat each subexpression formed with the product combinator, $*$, as a single transition.

Transition

$\ll$ τ `= t(0).a + p` $\rightarrow$ `t(0).a, t(0).i :=` τ, τ $\gg$

$*$ $\ll$ `valid(nexty0)` $\neq$ `valid(y(0))` $\rightarrow$ `y(0) := nexty0` $\gg$

The proof obligation is

$$\begin{array}{ll} & T(S_1) \wedge I(S_1) \wedge (\tau = t(0).a(S_1) + p) \\ \wedge & (\mathrm{valid}(\mathrm{nexty0}(S_1)) \neq \mathrm{valid}(y(0)(S_1))) \\ \wedge & (t(0).a(S_2) = \tau) \wedge (t(0).i(S_2) = \tau) \wedge (y(0)(S_2) = \mathrm{nexty0}) \\ \wedge & (\forall v \in V_P - \{t(0).a, t(0).i, y(0)\} : v(S_2) = v(S_1)) \\ \Rightarrow & I(S_2) \wedge T(S_2) \end{array}$$

Assume that the left side of the implication holds. The clauses of the left side of the implication will be referred to as L in the following arguments. Clauses $I.1$ through $I.6$ for $k > 0$ and clause $I.7$ hold in state S_2 because they held in S_1 and all variables in these clauses have the same values in S_1 and S_2. Noting that $\sigma(0) = 0$,

$$t(0).a(S_2) = t(0).i(S_2) = t(0).i(S_2) + \sigma(0) = \tau$$

which establishes clauses $I.1$, $I.2$, and $I.3$ in state S_2.

To show that clauses $I.4$, $I.5$, and $I.6$ hold in state S_2, it is shown below that `valid(y(0))` $\neq$ `valid(y(1))` and `t(0).i = t(1).i + p` in state S_2.

$$\begin{array}{rcll} t(0).a(S_1) & = & \tau - p & ,(L) \\ t(0).i(S_1) & = & \tau - p & ,(I.1(S_1)) \\ t(0).i(S_1) + \sigma(1) & < & \tau & ,(\delta(1) < p) \\ t(1).i(S_1) & > & t(0).i.(S_1) - p & ,(I.3(S_1), k = 1) \\ y(0).(S_1) & = & y(1).(S_1) & ,(I.6(S_1), k = 0) \\ t(0).i(S_1) & = & t(1).i.(S_1) & ,(I.5(S_1), k = 0) \\ t(0).i(S_1) & = & \tau - p & ,(\text{the above}) \\ \mathrm{valid}(\mathrm{nexty0}(S_1)) & \neq & \mathrm{valid}(y(0)(S_1)) & ,(L) \\ y(0)(S_2) & = & \mathrm{nexty0} & ,(L) \\ y(1)(S_2) & = & y(1)(S_1) & ,(L) \\ \mathrm{valid}(y(0)(S_2)) & \neq & \mathrm{valid}(y(1)(S_2)) & ,(\text{the above}) \\ t(0).i(S_2) & = & \tau & ,(L) \\ t(1).i(S_2) & = & t(0).i(S_1) & ,(L) \\ t(0).i(S_2) & = & t(1).i(S_2) + p & ,(\text{the above}) \end{array}$$

Therefore, I is maintained.

To show that T is maintained, it is noted that the only clauses of T that are affected by the transition from S_1 to S_2 are $\tau \leq t(0).a + p$, and

$$\begin{array}{l}(\text{valid}(y(0)) \neq \text{valid}(y(1))) \wedge (valid(y(1)) = \text{valid}(2)) \\ \quad \Rightarrow (\tau \leq \max(t(0).a, t(2).a) + d)\end{array}$$

As shown above, $t(0).i(S_2) = \tau$ which establishes both of these clauses.

Transitions

$\ll$ `nexty0.t, nexty0.f := FALSE, FALSE` $\gg$
$\|$ $\ll$ `nexty0.t, nexty0.f := TRUE, FALSE` $\gg$
$\|$ $\ll$ `nexty0.t, nexty0.f := FALSE, TRUE` $\gg$

The relation for each of these transitions includes the clause

$$\forall v \in V_P - \{\texttt{nexty0.t, nexty0.f}\} : v(S_2) = v(S_1)$$

All variables appearing in I or T have the same value in states S_1 and S_2. Therefore, I and T are trivially maintained by these transitions.

Transition

$\ll$ `(valid(y(j-1))` $\neq$ `valid(y(j))) AND (valid(y(j)) = valid(y(j+1)))`
  $\rightarrow$ `y(j) := y(j-1)`
$\gg$
$*$ $\ll$ $\tau \geq$ `max(t(j-1).a, t(j+1).a) + mind` $\rightarrow$ `t(j).a :=` τ $\gg$
$*$ $\ll$ `t(j).i := t(j-1).i` $\gg$

The proof obligation is

$$\begin{array}{ll} & T(S_1) \wedge I(S_1) \\ \wedge & (\text{valid}(y(j-1)(S_1)) \neq \text{valid}(y(j)(S_1))) \wedge (\text{valid}(y(j)(S_1)) = \text{valid}(y(j+1)(S_1))) \\ \wedge & (\tau \geq \max(t(j-1).a, t(j+1).a) + \text{mind}) \\ \wedge & (y(j)(S_2) = y(j-1)(S_1)) \wedge (t(j).a(S_2) = \tau) \wedge (t(j).i(S_2) = t(j-1).i(S_1)) \\ \wedge & (\forall v \in V_P - \{y(j), t(j).i, t(0).a\} : v(S_2) = v(S_1)) \\ \Rightarrow & I(S_2) \wedge T(S_2)\end{array}$$

Assume that the left side of the implication holds. The clauses of the left side of the implication will be referred to as L in the following arguments. Clauses $I.1$ and $I.7$, and clauses $I.2$ and $I.3$ for $k \neq j$, and clauses $I.4$, $I.5$, and $I.6$ for $k \notin \{j, j+1\}$ hold in state S_2 because they held in S_1 and all variables in these clauses have the same values in S_1 and S_2. Therefore, these clauses hold in S_2 also.

Establishing $I.2(S_2)$ uses several of the inequalities of I and T. The key idea is to derive upper bounds for $t(j-1).a$ and $t(j+1).a$ as functions of $t(j-1).i$. Then, the clause

$$\begin{array}{l}(\text{valid}(y(j-1)) \neq \text{valid}(y(j))) \wedge (\text{valid}(y(j)) = \text{valid}(y(j+1))) \\ \quad \Rightarrow (\tau \leq \max(t(j-1).a, t(j+1).a) + d)\end{array}$$

is used to derive a bound on τ and therefore $t(j).a(S_2)$.

$$\begin{array}{rcll}
t(j-1).a(S_1) & \leq & t(j-1).i.(S_1)+\sigma(j-1) & ,(I.2(S_1), k=j-1) \\
t(j-1).a.(S_1) & \leq & t(j-1).i.(S_1)+\sigma(j)-d & ,(\delta(j)\geq d) \\
\text{valid}(y(j-1)(S_1)) & \neq & \text{valid}(y(j)(S_1)) & ,(L) \\
t(j-1).i(S_1) & = & t(j).i(S_1)+p & ,(I.6(S_1), k=j-1) \\
\text{valid}(y(j)(S_1)) & = & \text{valid}(y(j+1)(S_1)) & ,(L) \\
y(j)(S_1) & = & y(j+1)(S_1) & ,(I.4(S_1)) \\
t(j+1).i(S_1) & = & t(j).i(S_1) & ,(I.5(S_1), k=j) \\
t(j+1).a(S_1) & \leq & t(j-1).i(S_1)-p+\sigma(j+1) & ,(I.2(S_1), k=j+1) \\
t(j+1).a(S_1) & \leq & t(j-1).i(S_1)+\sigma(j)-d & ,(\delta(j+1)\leq p-d) \\
\tau & \leq & t(j-1).i(S_1)+\sigma(j) & ,(T(S_1)) \\
t(j).a(S_2) & \leq & t(j-1).i(S_1)+\sigma(j) & ,(T(S_1)) \\
t(j).a(S_2) & \leq & t(j).i(S_2)+\sigma(j) & ,(L)
\end{array}$$

The last line is $I.2(S_2)$ for $k=j$.

$I.3(S_2)$ for $k=j$ follows directly from $I.3(S_1)$ for $k=j-1$.

$$\begin{array}{rcll}
\tau & \leq & t(j-1).i(S_1)+\sigma(j-1)+p & ,(I.3(S_1), k=j) \\
t(j).i(S_2) & = & t(j-1).i(S_1) & ,(L) \\
\sigma(j) & > & \sigma(j+1) & ,(\delta(j)>0) \\
\tau & \leq & t(j).i(S_2)+\sigma(j)+p &
\end{array}$$

The last line is $I.3(S_2)$ for $k=j$.

It follows directly from the consequences of L, that $y(j)(S_2)=y(j-1)(S_2)$, and $t(j).i(S_2)=t(j-1).i(S_2)$, which establishes $I.4$, $I.5$, and $I.6$ for $k=j-1$. Likewise, $\text{valid}(y(j)(S_2))\neq\text{valid}(y(j+1)(S_2))$, and $t(j).i(S_2)=t(j+1)(S_2)+p$, which establishes $I.4$, $I.5$, and $I.6$ for $k=j$.

The argument that T is maintained is analogous to the previous one for the transition that writes `y(0)`.

Transition

$\ll \tau$ `= t(n+1).a + p` $\gg$

* $\ll$ `y(n+1), t(n+1).a, t(n+1).i := y(n),` τ`, t(n).i` $\gg$

The proof obligation is

$$\begin{array}{ll}
& T(S_1) \wedge I(S_1) \\
\wedge & (\tau=t(n+1).a(S_1)+p) \wedge (y(n+1)(S_2)=y(n)(S_1)) \\
\wedge & (t(n+1).a(S_2)=\tau) \wedge (t(n+1).i(S_2)=t(n).i(S_1)) \\
\wedge & (\forall v\in V_P-\{y(n+1),t(n+1).i,t(n+1).a\}: v(S_2)=v(S_1)) \\
\Rightarrow & I(S_2) \wedge T(S_2)
\end{array}$$

Assume that the left side of the implication holds. The clauses of the left side of the implication will be referred to as L in the following arguments. Clause $I.1$, and clauses $I.2$ and

$I.3$ for $k < n+1$, and clauses $I.4$, $I.5$, and $I.6$ for $k < n$ hold in state S_2 because they held in S_1 and all variables in these clauses have the same values in S_1 and S_2. Therefore, these clauses hold in S_2 also.

By arguments analogous to those for the transition that writes `y(0)`, $t(n+1).a(S_2) = \tau$, $t(n+1).i(S_2) = t(n).i(S_2) = \tau - \sigma(n+1)$, and $y(n+1)(S_2) = y(n)(S_2)$. This establishes the remaining clauses of I.

The preceding arguments have shown that I is maintained by P. To complete the proof that I is an invariant, it must also be shown that any change of τ by the environment that satisfies the timing conditions also maintains I.

Assume otherwise; this means

$$\{I\}\ \tau \text{ changes } \{\neg I\}$$

It is shown below that this implies that the new value of τ violates one or more of the timing conditions.

Since τ only appears in clause $I.3$ of I, this clause must be violated. Choose k such that

$$\forall j : (0 \leq j \leq n+1) \Rightarrow (t(k).i + \sigma(k) \leq t(j).i + \sigma(j))$$

By assumption, $t(k).i + \sigma(k) < \tau$. There are three cases to consider:

$k = 0$:

This means that, $\tau > t(0).i + \sigma(0) + p$ which implies that $\tau > t(0).a + p$. A violation the timing condition $\tau \leq t(0).a + p$.

$1 \leq k \leq n$:

This means that $\tau > t(k).i + \sigma(k) + p$. By the choice of k,

$$t(k-1).i + \sigma(k-1) \geq t(k) + \sigma(k)$$

Since $\sigma(k-1) < \sigma(k)$, $t(k-1).i > t(k)$. Furthermore I and I held before changing τ, and changing τ changes no other variables; therefore,

$$\begin{array}{rcll}
y(k-1) & \neq & y(k) & ,(I.5, j = k-1) \\
\text{valid}(y(k-1)) & \neq & \text{valid}(y(k)) & ,(I.4, j = k-1) \\
t(k-1).i & = & t(k).i + p & ,(I.6, j = k-1) \\
t(k-1).a & \leq & t(k-1).i + \sigma(k-1) & ,(I.2, j = k) \\
t(k-1).a & \leq & t(k).i + \sigma(k) + p - d & ,(\delta(k) \geq d)
\end{array}$$

Similar arguments yield

$$\begin{array}{rrcl}
 & \text{valid}(y(k+1)) & = & \text{valid}(y(k)) \\
\text{and} & t(k+1).a & \leq & t(k).i + \sigma(k) + p - d
\end{array}$$

Applying the timing condition

$$\begin{array}{llcl}
(& (\text{valid}(y(k-1)) & \neq & \text{valid}(y(k))) \\
\quad \wedge & (\text{valid}(y(k)) & = & \text{valid}(y(k+1))) \\
) \ \Rightarrow & (\tau \leq \max(t(k-1).a, t(k+1).a) + d) & &
\end{array}$$

yields

$$\tau \leq t(k).i + \sigma(k) + p$$

A contradiction.

$k = n + 1$:
This means that $\tau > t(n+1).i + \sigma(n+1) + p$. Since $I.7$ held before changing τ, $t(n+1).a = t(n+1).i + \sigma(n+1)$. Therefore, $\tau > t(n+1).a + p$, which violates the timing condition $\tau \leq t(n+1).a + p$.

□ (I is an invariant.)

The correctness of STARI

I implies the two correctness conditions for STARI stated at the beginning of this section. Consider first $Q1$.

$$Q1 \equiv (\tau = t(0).a + p) \Rightarrow (y(1) = y(0))$$

By I,

$$\begin{array}{lcl}
(\tau = t(0).a + p) & & \\
\quad \Rightarrow & (t(0).i = \tau - p) & ,(I.1) \\
\quad \Rightarrow & (t(0).i + \sigma(1) < \tau) & ,(\sigma(1) < p) \\
\quad \Rightarrow & (t(1).i \neq t(0).i - p) & ,(I.3, k = 1) \\
\quad \Rightarrow & (y(1) = y(0)) & ,(I.6, k = 0)
\end{array}$$

as required.

Now consider $Q2$.

$$Q2 \equiv (\tau = t(n+1).a + p) \Rightarrow (\mathrm{valid}(y(n+1)) \neq \mathrm{valid}(y(n)))$$

By I,

$$\begin{array}{lrcll}
(\tau = t(n+1).a + p) & & & & \\
\quad \Rightarrow & (\tau & = & t(n+1).i + \sigma(n+1) + p) & ,(I.7) \\
\quad \Rightarrow & (\tau & > & t(n+1).i + \sigma(n) + p) & ,(\delta(n+1) > 0) \\
\quad \Rightarrow & (t(n).i & \neq & t(n+1).i) & ,(I.3, k = n) \\
\quad \Rightarrow & (y(n) & \neq & y(n+1)) & ,(I.5, k = n) \\
\quad \Rightarrow & (\mathrm{valid}(y(n)) & \neq & \mathrm{valid}(y(n+1))) & ,(I.4, k = n)
\end{array}$$

as required.