Automatic Verification of Asynchronous Circuits*

Trevor W. S. Lee   Mark R. Greenstreet   Carl-Johan Seger

Technical Report 93-40
Department of Computer Science
University of British Columbia
Vancouver, B.C. V6T 1Z4 Canada

November 12, 1993

Abstract

Asynchronous circuits are often used in interface circuitry where traditional, synchronous design methods are not applicable. However, the verification of asynchronous designs is difficult, because standard simulation techniques will often fail to reveal design errors that are only manifested under rare circumstances. In this paper, we show how asynchronous designs can be modeled as programs in the Synchronized Transitions language, and how this representation facilitates rigorous and efficient verification of the designs using ordered binary diagrams (OBDDs). We illustrate our approach with two examples: a novel design of a transition arbiter and a design of a toggle element from the literature. The arbiter design was derived by correcting an error in an earlier attempt. It is noteworthy that the error in the original design, found very quickly using the methods described in this paper, went unnoticed during more than 50 hours of CPU time simulating 2^{31} state transitions.

Keywords: hardware verification, self-timed designs, hardware description languages, arbiters, binary decision diagrams, invariants, model checking.

*This research was supported by operating grants OGPO 109688 and OGPO 138501 from the Natural Sciences Research Council of Canada and a fellowship from the Advanced Systems Institute.

1 Introduction

Speed-independent circuits operate without clocks and function correctly regardless of the internal delays of their components. Such designs are attractive due to their robustness, and are often used as interface logic between synchronous systems. However, their asynchronous nature makes design and verification difficult because standard simulation based techniques often fail to reveal errors that only show up under rare conditions. This motivates our research into verification tools that are more rigorous and more efficient than traditional simulation.

Our approach is based on ordered binary decision diagrams, (OBDDs) [Bry86]. Algorithms for manipulating Boolean functions represented by OBDDS are sufficiently efficient that these techniques can be applied to practical, real-world designs. The typical use of OBDDs in verifying safety properties is based on symbolic model checking[BCMD90]. In this approach, the behavior of a circuit is given by a state transition relation (represented as an OBDD), and properties of the circuit are verified by reachability analysis of this relation. Although OBDDs can represent complex next state relations, the relations for many designs are still much larger than can be handled efficiently.

Most approaches that have been proposed to alleviate the problem of large next state relations partition the relation into a collection of functions or smaller relations whose union is the complete relation. In general, it is tedious and difficult to find a good partitioning. Another approach is to recognize when the relation is in fact a function, since functions can be represented much more efficiently than general relations. As with partitioning, however, discovering that a relation is a function is often quite subtle and almost always very tedious.

Synchronized Transitions is a hardware description language in which digital circuits are modeled as parallel programs. Programs written in Synchronized Transitions describe both the computation and the structure of digital circuits and can be used to specify designs from very high levels of abstraction down to gate level descriptions. Synchronized Transitions has been used in the design of many integrated circuits including a vector-matrix multiplier [NSJ91] and a high-speed communications chip [Gre92]. Synchronized Transitions can be used to design traditional, synchronous systems [SG88] or to design speed-independent, asynchronous circuits[Sta93]. For either style of design, the syntactic structure of Synchronized Transitions programs provides the basis for partitioning the transition relation into a collection of next state functions. Thus, when a design is described using Synchronized Transitions, efficient, OBDD-based formal verification is possible.

In this paper we describe our translation of Synchronized Transitions programs into the FL language [Seg93], a functional language with built-in support for OBDDs. We then show how safety properties of the Synchronized Transitions program can be automatically verified using the translator and a library of predefined FL functions. In particular, we describe how program invariants can be checked and how to derive the weakest invariant sufficient to guarantee a desired safety property. We illustrate the applications of this approach with three examples. First, we verify that our arbiter guarantees mutual exclusion. Second, we show that all signals in this design obey a speed-independent protocol—basically, that there are no glitches. Finally, we use the ability of computing the weakest invariant to prove that one Synchronized Transitions program is a refinement (or implements) another Synchronized Transitions program, illustrating this approach with the arbiter and a toggle element.

2 Synchronized Transitions

Digital circuits are composed of signals, latches for storing these signals, and combinational logic for computing functions of signals. In a Synchronized Transitions program, state variables correspond to signals, and transitions model latches and combinational logic.

In this paper we will focus on a subset of Synchronized Transitions in which all variables are of Boolean or bounded integer subrange types, or of fixed array or record types constructed from these elementary types. This means that each variable has a finite set of possible values. Furthermore, the set of state variables of a Synchronized Transitions program is static and finite. Thus, the state space of a program is discrete and finite, and is given by the Cartesian product of the sets of possible values for each state variable. For a program, Q, we write V^Q to denote the set of all variables of Q and $\mathcal{Q}$ to denote the state space of Q (i.e. $\mathcal{Q}$ is the set of all assignments of values to variables of Q).

Property 1 *Let v be a variable declared to be of type T in a Synchronized Transitions program, Q. From a functional point of view, v is a function from states of Q to values of type T, i.e., $v : \mathcal{Q} \rightarrow T$.*

Transitions specify state changes using multi-assignments. For example,

$$\ll \texttt{a = b} \rightarrow \texttt{c := a} \gg$$

is a transition which specifies an update of `c` to the value of `a`. The update is only performed if the precondition, `a = b`, holds. This describes the operation of a Muller C element, a device that is frequently used in self-timed designs. Whenever the two inputs of a C element have the same value, the C element is enabled to set its output to this value. In general, a transition is of the form

$$\ll \textit{precondition} \rightarrow \textit{action} \gg$$

The precondition is a predicate over the system state that indicates when the transition may be performed. As a syntactic shorthand, a transition may be written without a precondition, in which case the transition is always enabled. The action is a multi-assignment which specifies the state transformation made by the transition; it can only be performed if the precondition holds. In a multi-assignment, all expressions on the right hand side are evaluated first, and then these values are assigned to the variables on the left hand side. All variables on the left hand side must be distinct.

Property 2 *Let $M \equiv l_1, l_2, \ldots l_m := r_1, r_2, \ldots r_m$ be the multi-assignment of a transition in a Synchronized Transitions program, Q. From a functional point of view, M is a function from $\mathcal{Q}$ to $\mathcal{Q}$ where*

$$\begin{array}{rcl} M(s_1) = s_2 & \Leftrightarrow & \forall i \in \{1 \ldots m\}.(l_i(s_2) = r_i(s_1)) \\ & \wedge & \forall v \in V^Q - \{l_1 \ldots l_m\}.(v(s_2) = v(s_1)) \end{array}$$

Property 3 *Let $t \equiv \ll G \rightarrow M \gg$ be a transition in a Synchronized Transitions program, Q. From a functional point of view, t is a partial function, f_t from states to states; i.e., $f_t : \{s \in \mathcal{Q} \mid G(s)\} \rightarrow \mathcal{Q}$, where*

$$f_t(s_1) = s_2 \quad \Leftrightarrow \quad G(s_1) \wedge (s_2 = M(s_1))$$

To describe any significant circuit, many transitions are needed. The asynchronous combinator, $\|$, combines two or more transitions which execute asynchronously, independently, and atomically. For example,

$$\ll \texttt{a<b} \rightarrow \texttt{a,b := b,a} \gg \; \| \; \ll \texttt{b<c} \rightarrow \texttt{b,c := c,b} \gg$$

specifies two independent transitions, each of which may be executed whenever its precondition is satisfied. This computation sorts `a`, `b`, and `c` into descending order. A program defines a set of transitions. This set is static and completely determined by the program text. An operational model for the execution of a program in Synchronized Transitions is repeated selection and execution of an arbitrary transition from this set.

Property 4 *Let $Q = \ll G_1 \rightarrow M_1 \gg \| \ll G_2 \rightarrow M_2 \gg \| \ldots \| \ll G_k \rightarrow M_k \gg$ be a Synchronized Transitions program. The program Q denotes a state transition relation, R^Q where*

$$(s_1, s_2) \in R^Q \quad \Leftrightarrow \quad \exists i \in \{1 \ldots k\}.(G_i(s_1) \wedge (s_2 = M_i(s_1)))$$

This property gives us a convenient decomposition of R^Q, the state transition relation for program Q. In particular, R^Q is the union of the relations corresponding to the partial state transition functions of each transition in Q.
To verify the correctness of programs, we employ two annotations:

`INITIALLY` Q_0: All possible initial states of the program satisfy the predicate Q_0.

`ALWAYS` I: If the program is executed starting from a state satisfying the `INITIALLY` annotation, then all reachable states satisfy I.

An `INITIALLY` annotation states an assumption about the program (describing, for example, the states reached when a "reset" signal is applied to the circuit). `ALWAYS` annotations are assertions about the program that must be verified. We describe the automatic verification of these assertions in Sections 5 and 6.

This section has presented the features of Synchronized Transitions that will be used in this paper. A more complete description of Synchronized Transitions including other combinators and mechanisms for structuring programs is presented in [SG90].

Synchronized Transitions differs from many other guarded command languages, e.g. CSP [Hoa78], because *shared variables* are used for communication instead of messages. This provides a very simple correspondence between programs and their hardware realizations. Synchronized Transitions also differs from typical hardware description languages, e.g., VHDL [HC86], Model [Eur], and ELLA [M+84], in that preconditions (guards) determine the temporal behavior. The language has no built-in clock concept or sequencing (";") constructs. By using simple primitives based on concurrent programming, Synchronized Transitions can be used effectively for synchronous [SG88], self-timed, and hybrid [Gre92] designs. In this paper, we only consider the use of Synchronized Transitions in the design and verification of self-timed circuits. There are strong similarities between Synchronized Transitions and UNITY, as developed by Chandy and Misra [CM88]. Both describe a computation as a collection of atomic conditional assignments without any explicit flow of control. Chandy and Misra propose this as a general programming paradigm.

3 An Example: a transition arbiter

The transition arbiter was originally described by Sutherland [Mol92] and is used as an example throughout this paper. The arbiter has two clients, each of which communicates with the arbiter using three signals: `r` (request), `g` (grant), and `d` (done). Transition signaling is employed; for example, if a client wants to request the privilege and its request signal, `r` is currently low (resp. high), the client makes the request by toggling the request signal to high (resp. low). Each client can be in one of four states described in the table below:

state description	name
$\mathtt{r} = \mathtt{g} = \mathtt{d}$	`Idle`
$\overline{\mathtt{r}} = \mathtt{g} = \mathtt{d}$	`Requesting`
$\overline{\mathtt{r}} = \overline{\mathtt{g}} = \mathtt{d}$	`Privileged`
$\mathtt{r} = \overline{\mathtt{g}} = \mathtt{d}$	`PendingDone`

When a client is in the `Idle` state, it may request the privilege by toggling the request signal, `r`, entering the `Requesting` state. The arbiter grants a pending request by toggling the grant signal, `g`, bringing the interface to the `Privileged` state. Then, the client can release the privilege by toggling the done signal, `d`, returning the interface to the `Idle` state from which the client can make another

```
TYPE ArbiterClient = RECORD
                      r, (* request, an arbiter input *)
                      g, (* grant,   an arbiter output *)
                      d, (* done:    an arbiter input *)
                        BOOLEAN;
                    END;

FUNCTION Privileged(c: ArbiterClient) =
BEGIN (c.g = c.r) AND (c.d ≠ c.r) END;

STATE
  c1, c2: ArbiterClient;

INITIALLY
  NOT (Privileged(c1) AND Privileged(c2));

ALWAYS
  NOT (Privileged(c1) AND Privileged(c2));

BEGIN
    (* transitions of the arbiter *)
      ≪ NOT Privileged(c2) → c1.g := c1.r ≫    (* grant client c1 *)
  ||  ≪ NOT Privileged(c1) → c2.g := c2.r ≫    (* grant client c2 *)

    (* transitions of client c1 *)
  ||  ≪ c1.r := NOT c1.g ≫                      (* request the privilege *)
  ||  ≪ c1.d := c1.g ≫                          (* release the privilege *)

    (* transitions of client c2 *)
  ||  ≪ c2.r := NOT c2.g ≫                      (* request the privilege *)
  ||  ≪ c2.d := c2.g ≫                          (* release the privilege *)
  END;
```

Figure 1: Synchronized Transitions program for a transition arbiter (specification)

request. After the arbiter grants the privilege, the client can toggle `d` and `r` to release the privilege and then request it again. Because the protocol is designed to be delay-insensitive [Udd84], these events may appear at the arbiter in either order. If a new request appears before the done from the previous cycle, the arbiter sees the interface in the `PendingDone` state. The arbiter has two clients, and a correct implementation will guarantee that both clients cannot be in the `Privileged` state at the same time.

The preceding paragraph gave a prose description of a transition arbiter. Although such prose can convey an intuitive understanding of the operation of the arbiter, imprecision of the English language preclude using such a description to rigorously verify a design. Instead, we will use a Synchronized Transitions program to describe the arbiter for our purposes of verification. Such a specification is given in Fig. 1. This program only states what a transition arbiter can do; it does not give a detailed description of a circuit that implements such an arbiter. An implementation is presented in Section 7. The `ALWAYS` annotation of the program in Fig. 1 asserts that the arbiter never enters a state in which both clients are granted the privilege. This is a safety property that we verify by model checking using OBDDs in Section 5.

4 FL and OBDDs

FL[Seg93] is a functional language similar to ML[RMH90]. As with ML, FL is strongly typed with a polymorphic type inference system. There is one feature of FL that makes it distinct from other functional languages: ordered binary decision diagrams (OBDDs) are first class objects in FL. In fact, every object of type `bool` is represented internally as an OBDD. Consequently, some rather unusual programming constructs can be executed efficiently. For example, in FL it is easy, and quite efficient, to execute a program that requires quantification over some set of Boolean variables. To give a simple example, consider verifying that $\forall a . \forall b.\ \ ((a \wedge \exists c.(b \vee c)) = \exists c.((a \wedge b) \vee (a \wedge c)))$ In FL this can be accomplished by simply evaluating the expression

```
!a. !b. ((a AND ?c. (b OR c)) = ?c. ((a AND b) OR (a AND c)));
```

In FL, `!` denotes a universal quantifier, and `?` denotes an existential quantifier. If the above expression is used as input to FL, the result is `T`, indicating that the expression is a tautology. Typically, significantly more complex expressions with more variables are used.

Originally, FL was designed as meta-language for one particular type of formal hardware verification, namely symbolic trajectory evaluation [SB93], but since FL is a general purpose functional language, it has become a language of choice for prototyping formal verification approaches that benefit from efficient handling of Boolean functions.

Translating ST Programs to FL Programs

In order to reason about Synchronized Transitions programs using the FL system, we have developed a simple translator that derives an internal FL representation for the corresponding Synchronized Transitions program. The translator is based on a Synchronized Transitions to C compiler that we had previously written. The translator produces an FL representation of the state transition relation corresponding to the Synchronized Transitions program, and a library provides functions to check various safety properties, such as those stated in an `ALWAYS` declaration.

Since the subset of Synchronized Transitions we support guarantees that the state space of the program is finite, the FL version of the program represents the state of the program as a list of Boolean variables. In fact, each element of type Boolean in the Synchronized Transitions program has a single corresponding Boolean variable in the FL representation and each finite subrange type is represented as a vector of Boolean variables of length $\lceil \lg |\mathcal{S}| \rceil$, where $|\mathcal{S}|$ denote the number of elements in the subrange.

In FL, the state transition relation is represented by a list of state transition functions; each function corresponds to a transition of the Synchronized Transitions program. These state transitions functions are in turn represented by tuples of the form (G, M), where G represents the precondition, a function from states to Booleans; and M represents the multi-assignment, a function that maps the state before performing the multi-assignment to the state produced by performing the multi-assignment.

For the interested reader, Appendix A lists the translated version of the Synchronized Transitions program of Fig. 1. Note that in FL the user can define new infix operators. This is used heavily in the translation to make it easier to read and understand.

5 Safety Conditions and Invariants

A system satisfies a *safety condition* P, if P is satisfied in all states reachable from any state satisfying the initial state predicate, Q_0. Establishing a safety condition involves two tasks: 1) verifying that the safety condition is satisfied in any initial state of the system, and 2) verifying that the safety condition holds in every state that can be reached from any of the initial states. A standard approach to verifying a safety property is to find an invariant, I, such that $Q_0 \Longrightarrow I$ and $I \Longrightarrow P$. Given a program Q and a predicate I over $\mathcal{Q}$, we say I is an *invariant* if and only if

$$\forall s. \forall s'.\ I(s) \wedge (s, s') \in R^Q \Longrightarrow I(s')$$

Intuitively, if I holds in a state, s, and I is an invariant, then I will hold in all immediate successors of s. Furthermore, I must hold in all states reachable from s as can be shown by a simple argument by induction. We write $inv(I, Q)$ to indicate that the predicate I is an invariant of the program Q.

To illustrate a simple invariant, consider the Synchronized Transitions program, Q with a single transition,

$$\ll \texttt{n := (n+2) MOD 6} \gg$$

Let I be the predicate "`n` is odd" (i.e. $(\texttt{n} \bmod 2) = 1$). It is easy to show that I is an invariant of Q as performing the transition when `n` is odd will lead to a new state where `n` is still odd.

In principle, in the FL version of an Synchronized Transitions program a proposed invariant I could be verified directly by computing (explicitly) the next state relation and checking

$$\forall s_1 \ldots s_n . \forall s'_1 \ldots s'_n .\ I(s_1 \ldots s_n) \wedge R(s_1 \ldots s_n, s'_1 \ldots s'_n) \Longrightarrow I(s'_1 \ldots s'_n)$$

where we have assumed that the s_i and s'_i are Boolean variables and that n variables are sufficient to represent the state of the Synchronized Transitions program. However, the next state relation is often much too large to be efficiently represented as an OBDD. Consequently, we partition the relation using Property 4 from Section 2 as justified by the following theorem:

Theorem 1 *Let* $Q = \ll G_1 \rightarrow M_1 \gg \| \ll G_2 \rightarrow M_2 \gg \| \cdots \| \ll G_k \rightarrow M_k \gg$ *be a Synchronized Transitions program. If* I *is a predicate over* $\mathcal{Q}$*, then*

$$inv(I,Q) \quad \Leftrightarrow \quad \forall i \in \{1 \ldots k\}.inv(I, \ll G_i \rightarrow M_i \gg)$$

Proof:

$$\begin{array}{rcll}
inv(I,Q) & = & \forall s \in \mathcal{Q}.\forall s' \in \mathcal{Q}.((I(s) \wedge (s,s') \in R^Q) \Longrightarrow I(s')) & \text{def. of invariant} \\
& = & \forall s \in \mathcal{Q}.\forall s' \in \mathcal{Q}. & \text{Property 4} \\
& & ((I(s) \wedge (\exists i \in \{1 \ldots k\}.(G_i(s) \wedge (s' = M_i(s))))) \Longrightarrow I(s')) & \\
& = & \forall i \in \{1 \ldots k\}.\forall s \in \mathcal{Q}.((I(s) \wedge G_i(s)) \Longrightarrow I(M_i(s))) & \text{predicate calculus} \\
& = & \forall i \in \{1 \ldots k\}.inv(I, \ll G_i \rightarrow M_i \gg) & \text{def. of invariant}
\end{array}$$

$\square$

This theorem allows us to verify an invariant by checking each transition in the Synchronized Transitions program separately.

To illustrate this approach, let Q be the Synchronized Transitions program for the arbiter shown in Fig. 1; let P be the the mutual exclusion property asserted in the `ALWAYS` declaration of the program,

$$\begin{array}{rcl}
P(\mathtt{c1,c2}) & = & \mathtt{NOT\ (Privileged(c1)\ AND\ Privileged(c2))} \\
& = & \neg((\mathtt{c1.g} = \mathtt{c1.r}) \wedge (\mathtt{c1.d} = \mathtt{c1.r}) \wedge (\mathtt{c2.g} = \mathtt{c2.r}) \wedge (\mathtt{c2.d} = \mathtt{c2.r}))
\end{array}$$

and let Q_0 be Q's initial state predicate, $Q_0 = P$. To verify that P is a safety property of Q, we must find an invariant, I, such that $Q_0 \Longrightarrow I \Longrightarrow P$. Because $Q_0 = P$, the only candidate for I is $I = P$. As described above, we verify that I is an invariant by considering each transition separately. For the transition

$$\ll \mathtt{NOT\ Privileged(c2)} \rightarrow \mathtt{c1.g\ :=\ c1.r} \gg$$

the proof obligation is

$$\begin{array}{l}
\forall\, \mathtt{c1.r}.\forall\, \mathtt{c1.g}.\forall\, \mathtt{c1.d}.\forall\, \mathtt{c2.r}.\forall\, \mathtt{c2.g}.\forall\, \mathtt{c2.d}. \\
\quad ((P(\mathtt{c1.r}, \mathtt{c1.g}, \mathtt{c1.d}, \mathtt{c2.r}, \mathtt{c2.g}, \mathtt{c2.d}) \wedge \neg\mathtt{Privileged(c2.r,c2.g,c2.d)}) \Longrightarrow \\
\quad P(M(\mathtt{c1.r}, \mathtt{c1.g}, \mathtt{c1.d}, \mathtt{c2.r}, \mathtt{c2.g}, \mathtt{c2.d})))
\end{array}$$

where

$$M(\mathtt{c1.r}, \mathtt{c1.g}, \mathtt{c1.d}, \mathtt{c2.r}, \mathtt{c2.g}, \mathtt{c2.d}) \quad = \quad (\mathtt{c1.r}, \mathtt{c1.r}, \mathtt{c1.d}, \mathtt{c2.r}, \mathtt{c2.g}, \mathtt{c2.d})$$

After some (rather tedious) manipulations, it is easy to see that the proof obligation indeed holds. Verifying the proof obligations for the other transitions are equally straightforward.

The proof obligations for verifying an invariant are readily expressed as an FL function, and the necessary FL code is generated directly from the Synchronized Transitions program using our `st2fl` translator. Note that since the state is represented as a list of Boolean variables and the functions $I()$, $G()$ and $M()$ are represented using OBDDs over these Boolean variables, it is not necessary to universally quantify over the state variables. Instead, we compute the Boolean expression $I^*(s_1, s_2, \ldots, s_n)$ defined as:

$$I^*(s_1, \ldots, s_n) \stackrel{\text{def}}{=} I(s_1, \ldots, s_n) \wedge G_i(s_1, \ldots, s_n) \Longrightarrow I(M_i(s_1, \ldots, s_n))$$

and determine whether this Boolean expression is a tautology. Note that computing $I(M_i(s_1, \ldots, s_n))$ involves substitution. In general, substitution can be an expensive OBDD operation; however, most multi-assignments in Synchronized Transitions programs are very simple, and substitution has never been a problem.

We note that safety properties are distinct from invariants. A predicate, I may be an invariant but not a safety property if it the initial state is not guaranteed to satisfy I (i.e. $Q_0 \not\Rightarrow I$). Conversely, a predicate P may be a safety property but not an invariant if there are states s and s' such that P holds in state s, s is unreachable from any state satisfying Q_0, and the program can make a state transition from state s to state s', (i.e. $(s, s') \in R^Q$), but P does not hold in state s'. A safety property is an assertion about all reachable states whereas the definition of an invariant considers all states of Q, not just the subset of states that are reachable. In general, the verification that a predicate is an invariant is simplified because it is not necessary to first construct the set of reachable states. Typically, we are interested in verifying safety properties, and, using the techniques described in this section, invariants are central to this verification. Manually finding a suitable invariant can be a difficult task. In the next section, we describe how invariants can be automatically derived.

6 Deriving Invariants

Given a program, Q, and a safety property, P, we are interested in finding the weakest invariant [Lam87] of Q that implies P. We write $win_{P,Q}$ to denote this invariant. When we say that $win_{P,Q}$ is the weakest invariant of Q that implies P, we mean that if I is any other invariant of Q that implies P, then if I is satisfied in some state s, $win_{P,Q}$ is satisfied in s as well. The existence of a weakest invariant can be seen by considering the set, $\mathcal{I}_{P,Q}$ of all invariants of Q that imply P. It is easily shown that this set is finite (because the state space of Q is finite) and non-empty (because "false" is an invariant of any program). If I_1 and I_2 are invariants, then $I_1 \vee I_2$ is an invariant as well. From these observations, it is straightforward to show that

$$win_{P,Q} \quad = \quad \bigvee_{I \in \mathcal{I}_{P,Q}} I$$

It is easy to verify that P is a safety property of Q if and only if $Q_0 \Longrightarrow win_{P,Q}$, i.e., if and only if the initial state predicate implies the weakest invariant.

We now derive our algorithm for finding $win_{P,Q}$. Let Q be a program, and let $\mathcal{Q}$ be the set of all states of Q. Let,

$$W_{P,Q} = \{s \in \mathcal{Q} \mid \forall t \in \mathcal{Q}.(s,t) \in R^* \Longrightarrow P(t)\}$$

where $R^* = \bigcup_{i=0}^{\infty} R^i$, is the reflexive and transitive closure of R. Let the predicate $I(s)$ be defined to hold for all s such that $s \in W_{P,Q}$. It is easy to show that I is an invariant of Q and that I implies P. Furthermore, if t is a state such that $\neg I(t)$, then there exists a state u such that u is reachable from t and $\neg P(u)$. Therefore, $I = win_{P,Q}$.

As noted earlier, it is often impractical to represent a state transition relation, (e.g. R) as an OBDD, and its transitive closure (e.g. R^*) may be even less suitable for OBDD representation. A more efficient approach calculates win as a fixed point computation. Let,

$$X_{P,Q}(i) = \{s \in \mathcal{Q} \mid \forall j \in \{0 \ldots i\}.\forall t \in \mathcal{Q}.((s,t) \in R^j \Longrightarrow P(t))\}$$

It is easy to show that $\lim_{i \to \infty} X_{P,Q}(i) = W_{P,Q}$. Furthermore, $X_{P,Q}(0) = \{s \in \mathcal{Q} \mid P(s)\}$, and for $i > 0$

$$\begin{aligned}
X_{P,Q}(i) &= \{s \in \mathcal{Q} \mid \forall j \in \{0 \ldots i\}.\forall t \in \mathcal{Q}.((s,t) \in R^j \Longrightarrow P(t))\} \\
&= X_{P,Q}(0) \cap \{s \in \mathcal{Q} \mid \forall j \in \{1 \ldots i\}.\forall t \in \mathcal{Q}.(((s,t) \in R^j) \Longrightarrow P(t))\} \\
&= X_{P,Q}(0) \cap \{s \in \mathcal{Q} \mid \forall t,u \in \mathcal{Q}.\forall j \in \{0 \ldots i-1\}.((((s,u) \in R) \wedge ((u,t) \in R^j)) \Longrightarrow P(t))\} \\
&= X_{P,Q}(0) \cap \{s \in \mathcal{Q} \mid \forall u \in \mathcal{Q}.((s,u) \in R) \Longrightarrow (u \in X_{P,Q}(i-1))\}
\end{aligned}$$

which yields the desired iteration. For all i, $X_{P,Q}(i+1) \subseteq X_{P,Q}(i)$, and $X_{P,Q}(i) \subseteq \mathcal{Q}$. Also, it is easy to see that if $X_{P,Q}(i+1) = X_{P,Q}(i)$ for some i, it follows that $X_{P,Q}(j) = X_{P,Q}(i)$ for all $j > i$. These facts, together with the fact that $\mid \mathcal{Q} \mid$ is finite, implies that the iteration converges to a fixed point in a finite number of steps. This fixed point is $W_{P,Q}$ as desired.

As in the previous section, we can improve the efficiency of this computation by decomposing the state transition relation, R, into the union of state transition functions using Property 4. We begin by computing $win_{P,Q}$ for the simple case where P consists of a single transition, $\ll G \rightarrow M \gg$. Let (see Fig. 2)

$$U(P) = P \wedge (G \Longrightarrow (P \circ M))$$

Consider a state, s, in which $U(P)$ does not hold. This means that either P does not hold in s, or that the transition is enabled in state s, but that performing the transition leads to a state that violates P. Clearly, s cannot satisfy $win_{P,Q}$ and we conclude $win_{P,Q} \Longrightarrow U(P)$. It is straightforward to show that $win_{P,Q} = win_{U(P),Q}$. Furthermore, if P is an invariant then $U(P) = P$; otherwise $U(P)$ is satisfied by fewer states then P. Generalizing, $win_{P,Q} = win_{U^i(P),Q}$ for any positive integer i, and either $U^i(P) = U^{i-1}(P)$ in which case $win_{P,Q} = U^i(P)$ or $U^i(P)$ is satisfied in fewer states than $U^{i-1}(P)$ which guarantees that this computation will eventually reach a fixed point. The sequence

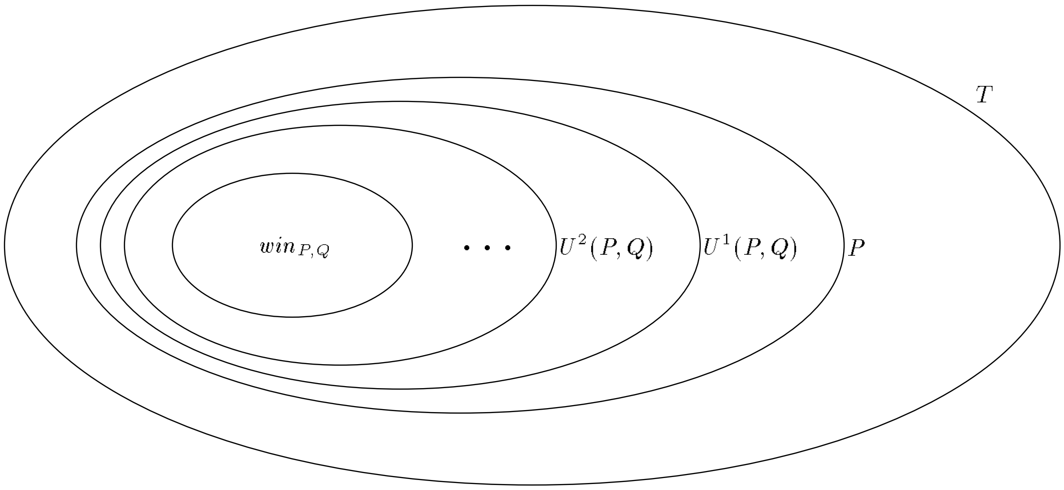


Figure 2: Computing the weakest invariant for a single transition.

```
// Compute win(P) of a single transition, << G -> M >>
let win1 P (G,M) s =
    letrec Pstar I s =
        let I' = (G AND I ==> (substitute (M(s)/s) I)) in
        if (I' == T) then I else (Pstar (I AND I') s) in
    Pstar P s;
```

Figure 3: FL program for computing the weakest invariant for a single transition.

of predicates produced by this method is illustrated in Fig 2, where each predicate is represented by the set of states that satisfy it. In FL, OBDDs are first-class citizens, providing an efficient implementation of higher-order functions (such as U). As shown in Fig. 3, a FL implementation for this algorithm to compute *win* of a single transition is extremely simple.

Using the `win1` function, the FL code for computing the the weakest invariant of a program that consists of a collection of transitions is given in Fig. 4. The basic idea is to iterate through the transitions and use the weakest invariant obtained from one transition as the starting approximation for the next iteration. Again, since we are always reducing the set of satisfying assignments, eventually we will reach a fixed point. This fixed point is the weakest invariant of the program that implies P. All operations needed to compute *win* are implemented efficiently as OBDD operations in FL. By construction, $win_{P,Q}$ is an invariant, and $win_{P,Q}$ implies Q as required. It remains to be shown that $Q_0 \Longrightarrow win_{P,Q}$ which, again, is readily tested using OBDDs. Many variations on this approach are possible. The version that we have presented here is simple and has worked quite

```
// Compute win of a list of transitions.
letrec win P trans_list s =
        letrec winlist I []         s = I /\
               winlist I (t:rest) s = winlist (win1 I t s) rest s in
        let P' = winlist P trans_list s in
        if (P == P') then P else (win P' trans_list s);
```

Figure 4: FL program for computing the weakest invariant that implies P for a Synchronized Transitions program.

well in practice.

7 An Implementation of the Transition Arbiter

In this section, we consider a speed-independent implementation of the transition arbiter that was specified in Section 1. The Synchronized Transitions program for this implementation is shown in Fig. 5. As indicated by the comments in the program, each transition corresponds to a simple gate, transparent latch, or Muller C-element, and Fig. 6 shows the corresponding schematic diagram. For simplicity, we do not show how the circuit is initialized to a state satisfying the `INITIALLY` section of the program.

This design was manually derived from the specification by choosing a transition that did not correspond to a simple circuit element and replacing the complex transition with a collection of simpler transitions that implement the same function. This process was repeated until the program of Fig. 5 was obtained. As the manual nature of this process provides many opportunities to introduce errors into the design, it is essential to verify that the final design has the desired properties. In this section, we first give an intuitive explanation of the operation of the speed-independent transition arbiter; we then describe how we verified the design using the `st2fl` program.

Operation of the arbiter

The arbiter is organized as three functional units: the internal arbiter, logic to interface the internal arbiter to the first client, and logic to interface the internal arbiter to the second client. The internal arbiter uses level signaling, and the interface circuits perform the necessary conversions between the transition signaling conventions of the clients and the level signaling of the internal arbiter.

The interface circuits are identical for each of the two clients, and each instance can be divided into two sub-units. The first sub-unit converts the level signaling output of the internal arbiter to the transition signaling conventions of the client. This sub-unit consists of the two transparent

```
STATE
  c1, c2:  ArbiterClient;
  s1, t1, u1, v1, w1, x1:  BOOLEAN;   (* internal variables for client c1 *)
  s2, t2, u2, v2, w2, x2:  BOOLEAN;   (* internal variables for client c2 *)

INITIALLY
  ( (NOT (c1.r OR c1.g OR c1.d OR c2.r OR c2.g OR c2.d))
    AND (NOT (s1 OR t1 OR u1 OR v1 OR x1)) AND (w1)
    AND (NOT (s2 OR t2 OR u2 OR v2 OR x2)) AND (w2)
  );

ALWAYS
  NOT (Privileged(c1) AND Privileged(c2));

  BEGIN
        (* The internal arbiter:  *)
        ≪ w1 := NOT (v1 AND w2) ≫   (* a NAND gate *)
    ||  ≪ w2 := NOT (v2 AND w1) ≫   (* a NAND gate *)
    ||  ≪ x1 := NOT w1 ≫            (* an inverter *)
    ||  ≪ x2 := NOT w2 ≫            (* an inverter *)

        (* Internal signals for client c1:  *)
    ||  ≪ NOT x1 → s1 := c1.r ≫    (* a transparent latch *)
    ||  ≪ t1 := s1 ≠ c1.d ≫         (* an XOR gate *)
    ||  ≪ u1 := s1 ≠ c1.g ≫         (* an XOR gate *)
    ||  ≪ t1 = u1 → v1 := t1 ≫     (* a C-element *)
    ||  ≪ x1 → c1.g := s1 ≫         (* a transparent latch *)

        (* Internal signals for client c2:  *)
    ||  ≪ NOT x2 → s2 := c2.r ≫    (* a transparent latch *)
    ||  ≪ t2 := s2 ≠ c2.d ≫         (* an XOR gate *)
    ||  ≪ u2 := s2 ≠ c2.g ≫         (* an XOR gate *)
    ||  ≪ t2 = u2 → v2 := t2 ≫     (* a C-element *)
    ||  ≪ x2 → c2.g := s2 ≫         (* a transparent latch *)

        (* actions of client c1:  *)
    ||  ≪ c1.r := NOT c1.g ≫        (* an inverter *)
    ||  ≪ c1.d := c1.g ≫            (* a buffer *)

        (* actions of client c2:  *)
    ||  ≪ c2.r := NOT c2.g ≫        (* an inverter *)
    ||  ≪ c2.d := c2.g ≫            (* a buffer *)
    END;
```

Figure 5: Synchronized Transitions program for a speed-independent transition arbiter

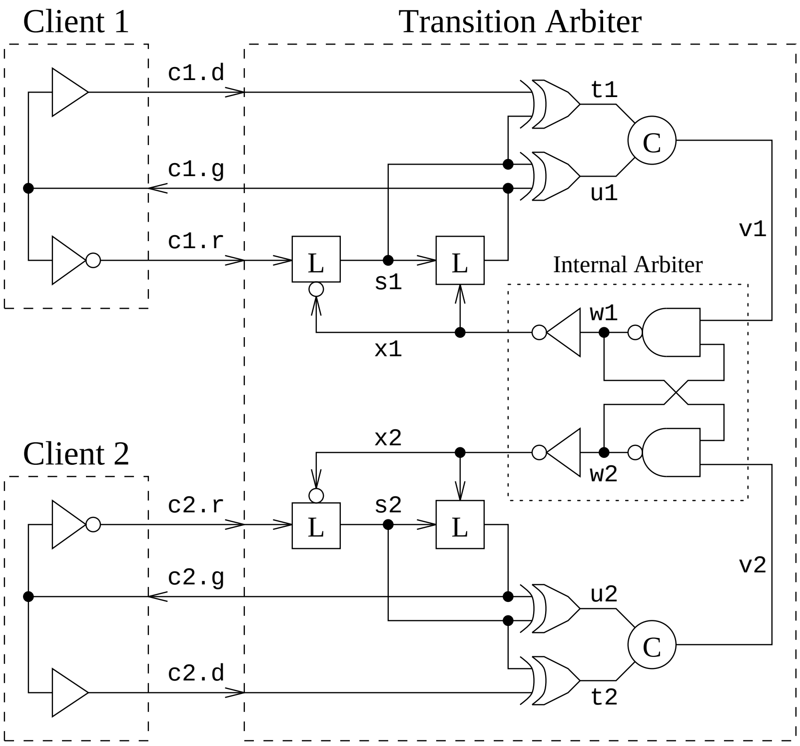


Figure 6: Schematic of the speed-independent transition arbiter.

latches that output the `s` and `c.g` signals. The transition arbiter grants a request by setting the grant signal to the same value as the request signal. This is done by these latches using a simple, two-step process: `c.r` is copied to `s` when the client does not have the internal privilege (i.e. `x` is false), and `s` is copied to `c.g` once the client acquires the internal privilege.

The second sub-unit converts from the transition signaling convention of the client to the level signaling of the internal arbiter. This sub-unit consists of the two exclusive-OR gates that output signals `t` and `u` and the Muller C-element that outputs the request to the internal arbiter, `v`. When the client is in the `Requesting` state, $\overline{\mathtt{r}} = \mathtt{g} = \mathtt{d}$, both `t` and `u` are both set to true, and this leads to a value of true for `v`, the request to the internal arbiter. Likewise, when the client is in the `Idle` state, `v` is set to false, releasing the internal privilege. Note that each client has two requesting states, one with `r` true and the other with `r` false; both lead to setting `v` true. Likewise, both `Idle` states lead to setting `v` false.

The preceding description has given an intuitive explanation to the function of each signal within the arbiter, and the ambitious reader may elaborate upon this by trying a few examples of clients requesting and releasing the privilege. The intuitive approach is helpful when designing a circuit such as this arbiter; however, it does not prove correct operation for all possible inputs or internal delays. We next present a rigorous verification using `st2fl`.

Verification of the arbiter

The safety property, P, that we require of the speed-independent arbiter shown in Fig. 5 is the same as the one stated in the specification (Fig. 1). However, the verification of P is more involved for the speed-independent version. In particular, P *is not* an invariant of the program for the speed-independent arbiter. The predicate P only constrains the values of interface variables, requiring that both clients cannot simultaneously have the privilege. An invariant that will imply P must also constrain the internal state of the arbiter to exclude, for example, states in which P is satisfied but a transition is enabled whose execution would violate P. For example, the state

```
  c1.r, c1.g, c1.d,  s1, t1, u1, v1, w1, x1,  c2.r, c2.g, c2.d,  s2, t2, u2, v2, w2, x2
= T,    T,    F,     T,  T,  T,  T,  F,  T,   T,    F,    F,     T,  F,  F,  F,  T,  T
```

is a state in which `Privileged(c1)` and `NOT Privileged(c2)` hold, thus P is satisfied. Performing the transition,

$$\ll \mathtt{x2} \rightarrow \mathtt{c2.g} := \mathtt{s2} \gg$$

leads to the state

```
  c1.r, c1.g, c1.d,  s1, t1, u1, v1, w1, x1,  c2.r, c2.g, c2.d,  s2, t2, u2, v2, w2, x2
= T,    T,    F,     T,  T,  T,  T,  F,  T,   T,    F,    F,     T,  F,  F,  F,  T,  T
```

which violates P.

For the sake of comparison, we manually derived a suitable invariant. This invariant is a complicated Boolean expression with more than two hundred literals! It is a time consuming task to derive such an invariant which, without the automatic tools described in this paper, must be repeated each time the design is modified. Furthermore, verifying that this expression is indeed an invariant and that the necessary implications are satisfied would involve more tedious and error prone effort. Using only manual methods, it would be difficult to produce a compelling argument for the correctness of this relatively simple design of only fourteen gates and latches.

Although an expression with several hundred literals may be unwieldy for humans, such expressions can be readily derived and manipulated using OBDDs. Using the `st2fl` program and the FL function for automatically computing *win*, we verified mutual exclusion for the speed-independent arbiter in under fifteen seconds of CPU time on a SPARC-10 workstation.

Debugging designs with `st2fl`

In the previous examples, we have shown how `st2fl` can be used to verify safety properties of designs, and our examples showed correct designs. An equally, if not more, important application of these methods is to reveal design errors. VLSI design is an iterative process; with each iteration, the designer corrects errors found in the previous version or adds functionality required by the complete system. Finding errors as soon as possible can make a significant contribution to the designer's productivity.

As an example, we consider an incorrect design of a speed-independent transition arbiter that we tried before we arrived at the design shown in Fig. 6. This design is shown in Fig. 7. The intended operation of this design is similar to the that of the correct design shown if Fig. 6. The central difference is that the requests to the internal arbiter (`v1` and `v2`) are generated by a transparent latch instead of a C-element. "Correct" operation requires that the latch be disabled before a grant is issued. This is the purpose of the exclusive-or gates that generate `t1`, `t2`, `z1`, and `z2`. Note that when, for example, signal `c1.d` changes the exclusive-or gate that outputs `z1` is excited. However, the gate that outputs `t1` is excited as well, and changing `t1` disables the output transition on `z1`. Our verification tools revealed this race and showed that this circuit does not guarantee mutual exclusion when arbitrary gate delays are admitted.

When we first designed this circuit, we simulated it for 500,000 state transitions by compiling its Synchronized Transitions program to C, and then compiling and executing the C program. Mutual exclusion was maintained throughout the simulation. We then attempted to verify mutual exclusion

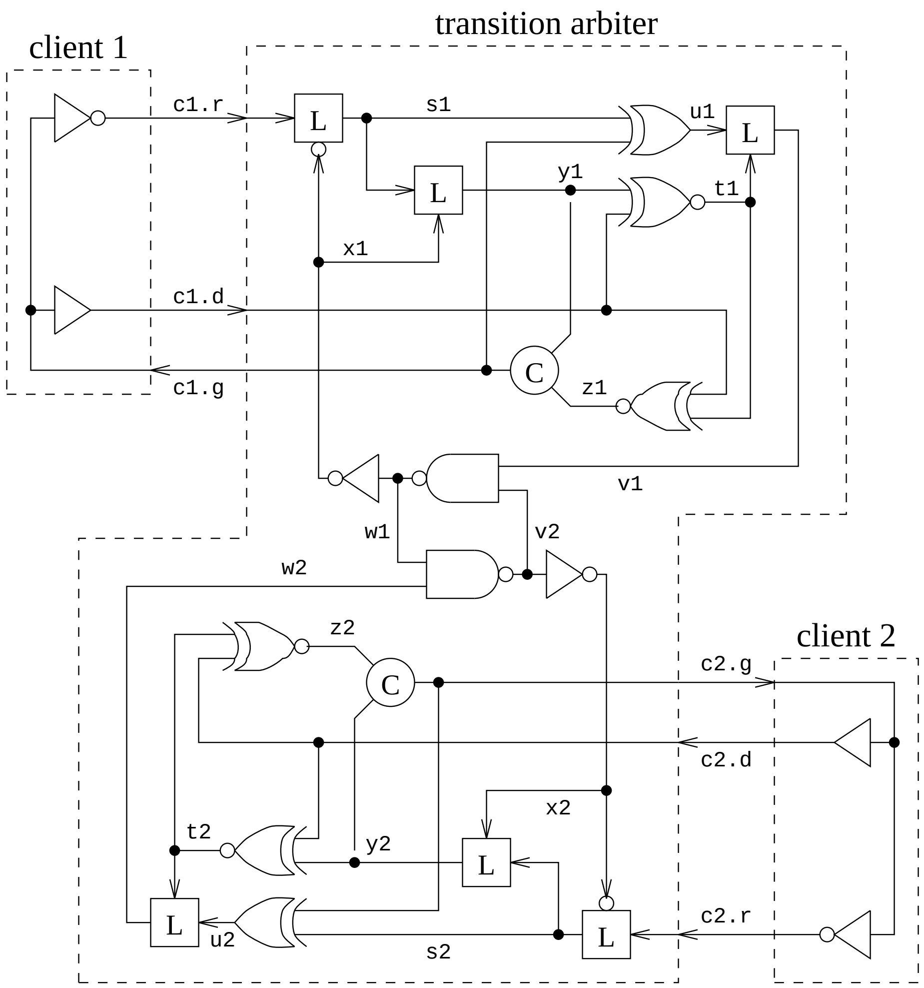


Figure 7: An incorrect implementation of a speed-independent transition arbiter.

as a safety property using our `st2fl` program. Within one minute of execution, FL reported that the weakest invariant for mutual exclusion was "false," i.e. from any state, a state could be reached that violated mutual exclusion. Furthermore, the program gave a shortest sequence of transitions that starts in a state satisfying the `INITIAL` predicate, but leads to a state that violates the `ALWAYS` predicate. This trace consisted of 38 transitions.

Given this experience, we were curious as to how much simulation would be required to reveal the error. The Synchronized Transitions language includes non-deterministic selection of transitions when more than one is enabled, and our simulator chooses among concurrently enabled transitions with equal probability. Perhaps a longer simulation would reveal the error. Our simulation included a counter of the number of state transitions performed. After 50 CPU hours and 2^{31} state transitions, the counter overflowed, ending the simulation, without a single violation of mutual exclusion encountered.

8 Other Verification Tasks

In this section we will illustrate how other verification tasks can be phrased in terms of computing weakest invariants of various predicates. In particular, we will illustrate how checking for hazard-freedom and refinement can be accomplished.

8.1 Speed-Independence

In the previous section, we presented a speed-independent implementation of the transition arbiter that was specified in Section 1. A design is speed-independent if

1. it functions correctly independent of the speeds of its components;
2. once a component is excited to change its output, it remains excited until the change is completed[1].

Both requirements for a correct speed-independent circuit can be verified using the methods described in the previous sections of this paper. When several transitions are enabled, the semantics of the || combinator allows a non-deterministic selection of which transition to execute. This corresponds to an arbitrary delay between the excitation of a circuit element and the corresponding change of its output. Thus, showing that a program satisfies its specification establishes that the corresponding circuit functions correctly independent of the speeds of its components. For the transition arbiter, we must show that the program can never grant both clients simultaneously.

[1]The second condition is violated by the NAND gates of the internal arbiter described in the previous section. We regard the internal arbiter as a single component and note that care must be taken in the implementation of this component [CM73]. Our design is based on that of Seitz [Sei79].

In our approach, each component of the circuit is modeled by a corresponding transition in the Synchronized Transitions program. In this case, the second requirement for speed-independence is satisfied if there are no "interfering" transitions. Intuitively, two transitions interfere in state s if both can be enabled from the same state, s, performing one of the transitions from state s modifies values read by the other. Let $G_t(s)$ denote that the precondition of transition t is satisfied in state s, let $W_t(s)$ denote the set of state variables that are modified (i.e. written) when performing t in state s, let R_t denote the set of state variable read by t. We write $SI_{t_1,t_2}(s)$ to denote the predicate that indicates the states in which t_1 does not interfere with t_2:

$$SI_{t_1,t_2}(s) \quad = \quad (G_{t_1}(s) \wedge G_{t_2}(s)) \Rightarrow (W_{t_1}(s) \cap R_{t_2} = \emptyset)$$

We note that $SI_{t_1,t_2}(s)$ is an ordinary predicate over states that can be derived directly from the Synchronized Transitions representations of t_1 and t_2. For a program, P, and a state s, $SI_P(s)$ denotes that there are no pairs of transitions in P that interfere in state s. If a program is of the form $t_1 \| t_2 \| \ldots \| t_n$, then

$$SI_P(s) = \bigwedge_{\substack{1 \leq i,j < n \\ i \neq j}} SI_{t_i,t_j}(s)$$

We note that $SI_P(s)$ is an ordinary predicate over states that can be derived directly from the program text. To verify the second requirement for speed independence, `st2fl` derives a definition of SI_P directly from the program text along with the assertion that $Q_0 \Rightarrow win_P(SI_P)$.

When we apply this test for speed-independence to the Synchronized Transitions program for the arbiter shown in figure 5, it reports, as expected, that the transitions for the two NAND gates interfere, but that there are no other interfering pairs of transitions. Alternatively, we can annotate the program to indicate that we know that the NAND gates interfere with each other; in which case, the design is verified with no errors reported.

8.2 Refinement

In the previous sections, we presented two programs for a transition arbiter. The first program (see Fig. 1) was a rather abstract description that specified the interaction of the arbiter with its clients. The second program (Fig. 5) described a gate-level implementation of this arbiter. We showed how our techniques could be used to verify that each of these programs ensured mutual exclusion. In this subsection, we present another approach to this verification task. In particular, we show how to verify that the gate-level description of a circuit implements a more abstract specification. To say that a circuit is an implementation of a specification means that everything that the circuit does corresponds to a behavior allowed by the specification. As a consequence, the circuit inherits all safety properties of the specification, and we don't have to repeat the verification of these assertions for the circuit. In the case of the arbiter, having verified that the specification ensures mutual exclusion, we can verify that the circuit is an implementation of the specification and thereby ensure that the circuit guarantees mutual exclusion as well. In the remainder of this subsection, we give a more precise definition of refinement and show how refinement of Synchronized Transitions programs can be verified using OBDDs. Initially, we use the transition arbiter as an example, and then we show how the same techniques can be applied to the verification of a toggle element.

To define refinement, we first examine what it means for an action of an implementation to correspond to an action of a specification. An action of a Synchronized Transitions program is a state transition and can be represented by a pair of states, (s_1, s_2) indicating that an execution of the program can produce state s_2 from s_1 in a single step. A state assigns a value to each state variable of the program. Typically, a specification and an implementation will have different sets of state variables, and states of the two programs are, therefore, not directly comparable. For example, the implementation of the arbiter given in Fig. 5 has state variables `s1` ...`x1` and `s2` ...`x2` that do not appear in the specification.

To compare states of two programs with different state variables, an abstraction function must be defined. This function maps states of the implementation program to states of the specification program. Often, this function is very simple; for example, the abstraction function for the arbiter programs maps the values of the interface variables of the circuit (i.e. `r1`, `g1`, `d1`, `r2`, `g2`, and `d2`) to the corresponding variables of the specification and discards the values of the internal variables (i.e. `s1` ...`x1` and `s2` ...`x2`). In other cases, the representation of a variable may be changed; for example, the specification may include an integer valued variable that is represented by an array of Boolean variables in the implementation; in such cases, a non-trivial abstraction function may be necessary.

We can now give a precise statement of what it means for one program to be an implementation of another. Given a specification program, Q, an implementation program, $\widetilde{Q}$, and an abstraction function A, let R^Q denote the state transition relation of Q and $R^{\widetilde{Q}}$ denote the state transition relation of $\widetilde{Q}$. $\widetilde{Q}$ is a refinement of Q if an only if for every pair of states, $(\tilde{s}_1, \tilde{s}_2)$ in $R^{\widetilde{Q}}$, either

$(A(\tilde{s}_1), A(\tilde{s}_2))$ is in R^Q

or $A(\tilde{s}_1) = A(\tilde{s}_2)$

In the first case, the action of the implementation corresponds directly to an action of the specification; in the second case, both states of the implementation map to the same state of the specification, and this action is called a "stuttering" action. Stuttering actions occur when the state transition in the implementation only alters the value of internal variables that do not correspond to any variables of the specification.

Although the preceding definition refers to the state transition relations of both the specification program, Q, and the implementation, $\widetilde{Q}$, it is not necessary to represent these relations explicitly, and significant performance improvement are achieved for automatic verification when an implicit representation is used. Our approach is similar to the one we used for verifying invariants and safety properties. The following derivation shows how this is achieved.

Let $Q = \ll G_1 \rightarrow M_1 \gg \| \ll G_2 \rightarrow M_2 \gg \| \ldots \| \ll G_m \rightarrow M_m \gg$

$\widetilde{Q} = \ll \widetilde{G}_1 \rightarrow \widetilde{M}_1 \gg \| \ll \widetilde{G}_2 \rightarrow \widetilde{M}_2 \gg \| \ldots \| \ll \widetilde{G}_n \rightarrow \widetilde{M}_n \gg$

$S^{\widetilde{Q}}$ = the state space of $\widetilde{Q}$

Then $\widetilde{Q}$ implements Q iff

$$\begin{aligned}
& \forall(\tilde{s}_1, \tilde{s}_2) \in R^{\widetilde{Q}}.(A(\tilde{s}_1) = A(\tilde{s}_2)) \vee ((A(\tilde{s}_1), A(\tilde{s}_2)) \in R^{Q}) \\
\equiv\ & \forall(\tilde{s}_1, \tilde{s}_2) \in S^{\widetilde{Q}} \times S^{\widetilde{Q}}.(\exists j \in \{1 \ldots n\}.\widetilde{G}_j(\tilde{s}_1) \wedge (\tilde{s}_2 = \widetilde{M}_j(\tilde{s}_1))) \Longrightarrow \\
& \qquad (A(\tilde{s}_1) = A(\tilde{s}_2)) \vee (\exists i \in \{1 \ldots m\}.G_i(A(\tilde{s}_1)) \wedge (A(\tilde{s}_2) = M_i(A(\tilde{s}_1)))) \\
\equiv\ & \forall \tilde{s} \in S^{\widetilde{Q}}.\forall j \in \{1 \ldots n\}.\widetilde{G}_j(\tilde{s}) \Longrightarrow \\
& \qquad (\widetilde{M}_j(\tilde{s}) = \tilde{s}) \vee (\exists i \in \{1 \ldots m\}.G_i(A(\tilde{s}_1)) \wedge (A(\widetilde{M}_j(\tilde{s}_1)) = M_i(A(\tilde{s}_1))))
\end{aligned}$$

The first equivalence is the definition of refinement from the previous paragraph, and the last line gives an equivalent expression that can be efficiently checked using OBDDs. In particular, we verify that the expression

$$\begin{aligned}
\Theta(\tilde{s}) = \forall j \in \{1 \ldots n\} & \\
& (\widetilde{M}_j(\tilde{s}) = \tilde{s}) \vee (\exists i \in \{1 \ldots m\}.G_i(A(\tilde{s}_1)) \wedge (A(\widetilde{M}_j(\tilde{s}_1)) = M_i(A(\tilde{s}_1))))
\end{aligned}$$

is a tautology.

The tests for refinement described above are very strict. In particular, all actions admitted by the state transition relation of the implementation must correspond to actions of the specification, *even if the state from which the transition is performed is unreachable.* Often, it is reasonable to implement a specification with a circuit that functions correctly as long as the environment is well-behaved and the system remains in the intended region of operation; if the environment does something forbidden by the specification, then the circuit is no longer required to function "correctly." This notion can be captured by an invariant that describes the set of states that can be reached under correct operation. To verify this invariant, the actions of the environment must be considered along with those of the circuits. This is why we included transitions to model the actions of the clients in our programs for the specification and implementation of the transition arbiter (a more complete an rigorous treatment of these issues can be found in [AL89]).

We note that the predicate Θ as defined above identifies the states of $\widetilde{Q}$ from which any state transition corresponds to an action of Q. Thus, if Θ is a safety property of $\widetilde{Q}$, then $\widetilde{Q}$ is a refinement of Q when only reachable states are considered. Given the specification program, Q, an implementation program, $\widetilde{Q}$, and an abstraction function A all written in the Synchronized Transitions language, our translator automatically produces the predicate Θ and produces FL code to verify $\widetilde{Q}_0 \Rightarrow win(\Theta, \widetilde{Q})$ as required, where $\widetilde{Q}_0$ is the initial state predicate for $\widetilde{Q}$.

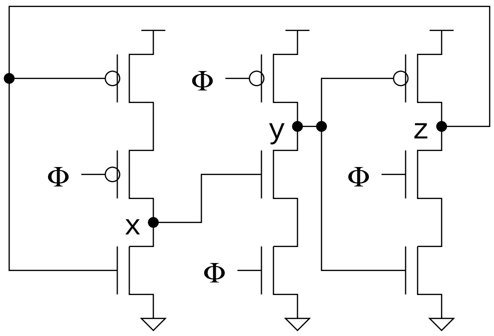
Using these methods, we have verified that the gate-level description of the arbiter (Fig. 5) is a refinement of the specification program (Fig. 1). The verification was performed automatically, and the CPU time required was nearly identical to that of verifying mutual exclusion for the gate-level program.

A toggle element

To show the applications of these methods to a transistor level design, we consider Yuan and Svensson's toggle element [YS89]. The Synchronized Transitions specification of a toggle element is straightforward as shown below:

```
   ≪ phi → q1 := q0 ≫
|| ≪ NOT phi → q0 := NOT q1 ≫
|| ≪ stable() → phi := NOT phi ≫
```

Where `stable()` = $(\texttt{phi} \Longrightarrow (q1 = q0)) \wedge (\neg\texttt{phi} \Longrightarrow (q1 \neq q0))$. The first two transitions are a two-phase description of the behavior of the flip-flop. The third transition states that the clock can change when the transitions that modify `q0` and `q1` have been performed.



Φ	x	y	z
L	L	H	H
H	L	H	L
L	H	H	L
H	L	L	H
L	L	H	H

Figure 8: Yuan-Svensson toggle element

Fig. 8 shows the implementation of a toggle element proposed by Yuan and Svensson. This design employs precharged logic and dynamic storage along with a single-phase clocking scheme to achieve operation at very high clock frequencies. In Synchronized Transitions, this toggle element can be described by giving next state functions for each of the signals `x`, `y`, `z`, and `q`, and writing a separate transition to update each of these signals according to these functions. For example, the next state function and the transition for `x` are

```
nextx(phi, x, y, z) = (NOT z) AND ((NOT phi) OR x);
≪ x := nextx(phi, x, y, z) ≫
```

The abstraction mapping is given by stating for each variable of the specification, an expression of variables in the implementation. If a variable has the same name in both the specification and implementation (e.g. `phi`), then that variable is assumed to have the same value in both programs as well. By examining the behavior of the toggle element as shown in Fig. 8, we choose the abstraction mapping:

```
q0 = x OR NOT y;        q1 = z AND NOT x;
```

```
  STATE
    x, y, z:  BOOLEAN; (* signals output by the Yuan-Svensson toggle element *)
    phi:  BOOLEAN; (* the clock *)
  INITIALLY (NOT x) y AND AND z;
  IMPLEMENTS (* specification of toggle element *)
    STATE
      q0:  BOOLEAN = x OR NOT y;
      q1:  BOOLEAN = z AND NOT x;
      phi:  BOOLEAN; (* implicitly mapped to phi in implementation *)
    FUNCTION stable() = ((q0 = q1) = phi); (* q0 and q1 have settled *)
  BEGIN
      ≪ phi → q1 := q0 ≫
    ‖ ≪ NOT phi → q0 := NOT q1 ≫
    ‖ ≪ stable() → phi := NOT phi ≫
  END;
  FUNCTION nextx(phi, x, y, z) = (NOT z) AND ((NOT phi) OR x);
  FUNCTION nexty(phi, x, y, z) = (NOT phi) OR ((NOT x) AND y);
  FUNCTION nextz(phi, x, y, z) = (NOT y) OR ((NOT phi) AND z);
BEGIN
    ≪ x := nextx(phi, x, y, z) ≫
  ‖ ≪ y := nexty(phi, x, y, z) ≫
  ‖ ≪ z := nextz(phi, x, y, z) ≫
  ‖ ≪ q := nextq(phi, x, y, z) ≫
  ‖ ≪  (x = nextx(phi, x, y, z)) AND
        (y = nexty(phi, x, y, z)) AND
        (z = nextz(phi, x, y, z)) →
        phi := NOT phi ≫
END;
```

Figure 9: Synchronized Transitions program of a Yuan-Svensson toggle element.

Fig. 9 shows the Synchronized Transitions program for the toggle element. The precondition of the transition that modifies `phi` is obtained by applying the inverse of the abstraction mapping to the precondition of the corresponding transition in the specification. The clauses state how the `x`, `y`, and `z` signals must settle before the next change of the `phi` input. These are timing assumptions that must be checked separately. Using our `st2fl` translator, the verification that the Yuan-Svensson design implements its specification takes less than one second.

9 Conclusion

We have shown how ordered binary decision diagrams (OBDDs) can be used to verify safety properties of speed-independent, asynchronous circuits described as programs written in the Synchronized Transitions language. In particular, we can verify that a particular predicate is an invariant or find the weakest invariant of a program that implies as desired safety property. A program written in Synchronized Transitions defines a state transition relation; however, this relation has a simple, syntactic decomposition into state transition functions. With this decomposition, Synchronized Transitions programs can be efficiently represented as OBDDs, which makes our methods applicable to practical designs.

We have illustrated our approach by considering three programs describing an arbiter. The first program is a simple, high-level specification of the arbiter; it guarantees mutual exclusion and describes the communication protocol between the arbiter and its clients. For this program, we verified that mutual-exclusion is an invariant of the program and that this invariant is satisfied by any allowed initial state. The second program is an implementation of the arbiter. For this program, a non-trivial invariant that guarantees mutual exclusion is much more complicated, and we described how this is automatically generated by our software. Finally, we presented an incorrect implementation of the arbiter and described an error that was quickly located by our software. Although the circuit was relatively simple (22 "gates"), the error was not revealed by a simulation of over two billion state transitions. We believe that these examples demonstrate the utility of formal methods for VLSI design.

This paper describes ongoing research and there are many areas that we are interested in exploring. We plan to investigate verifying liveness properties using the partitioned state transition relations that Synchronized Transitions offers. Other issues include representing infinite domains (e.g. integers) with OBDDs, supporting the other combinators of Synchronized Transitions (e.g. those used to describe synchronous circuits), and hierarchical verification based on refinements.

References

[AL89] Martín Abadi and Leslie Lamport. Composing specifications. In J.W. de Bakker et al., editors, *Proceedings of the REX Workshop, "Stepwise Refinement of Distributed Systems"*. Springer-Verlag, 1989. LNCS 430.

[BCMD90] J. R. Burch, E.M. Clarke, K.L. McMillan, and D.L. Dill. Sequential circuit verification using symbolic model checking. In *Proceedings of the 27th Design Automation Conference*. ACM, 1990.

[Bry86] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, Aug. 1986.

[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.

[Eur] European Silicon Structures, Bracknell, Berkshire RG12 3DY, United Kingdom. *Designing in MODEL*, doc. no. ES2-014-0055 edition. SOLO Reference Manual.

[Gre92] Mark R. Greenstreet. Using Synchronized Transitions for simulation and timing verification. In Jørgen Staunstrup and Robin Sharp, editors, *1992 Workshop on Designing Correct Circuits*, pages 215–236, Lyngby, Denmark, January 1992. Elsevier. An earlier version published as Matsushita Information Technology Laboratory technical report MITL-TR-01-91.

[HC86] D.D. Hill and D.R. Coelho. *Multi-level Simulation for VLSI Design.* Kluwer Academic Publishers, 1986.

[Hoa78] C.A.R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, August 1978.

[Lam87] Leslie Lamport. *win* and *sin*: Predicate transformers for concurrency. Technical Report 17, Digital Equipment Corporation, Systems Research Center, Palo Alto, CA, May 1987.

[M^+84] J.D. Morison et al. ELLA: Hardware description or specification. In *Proceedings of 1984 IEEE ICCAD*, 1984.

[Mol92] C. Molnar. Personal communications. 1992.

[NSJ91] Christian D. Nielsen, Jørgen Staunstrup, and Simon R. Jones. A delay-insensitive neural network engine. In J.G. Delgado-Frias and W.R. Moore, editors, *Proceedings of Workshop on VLSI for Artificial Intelligence and Neural Networks.* Plenum Press, 1991.

[RMH90] M. Tofte R. Milner and R. Harper. *The Definition of Standard ML*. MIT Press, Boston, MA., U.S.A., 1990.

[SB93] C-J. Seger and R. E. Bryant. Formal verification by symbolic evaluation of partially-ordered trajectories. Technical Report UBC-CS-93-8, Department of Computer Science, University of British Columbia, Vancouver, B.C., Canada, 1993.

[Seg93] C-J. Seger. Voss—a formal hardware verification system, user's guide. Unpublished Manuscript, Oct. 1993.

[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.

[Sta93] Jørgen Staunstrup. *A Formal Approach to Hardware Design*. Kluwer, 1993. in press.

[Udd84] Jan T. Udding. *Classification and Composition of Delay-Insensitive Circuits*. PhD thesis, Eindhoven University of Technology, 1984.

[YS89] Jiren Yuan and Christer Svensson. High-speed CMOS circuit technique. *IEEE Journal of Solid-State Circuits*, 24(1):62–70, February 1989.

Appendix A—FL program derived from Synchronized Transitions specification.

```
let ArbiterClient_ = st_rec_b [
   ("r", BOOLEAN_),
   ("g", BOOLEAN_),
   ("d", BOOLEAN_) ] [];
let ArbiterClient s = st_var (s, ArbiterClient_);

letrec privileged c =
   ((c:-"g") is (c:-"r")) and ((c:-"d") is_not (c:-"r"));

let c1_ = ArbiterClient "c1";
let c1 = st_state c1_;

let c2_ = ArbiterClient "c2";
let c2 = st_state c2_;

let st_vl = [c1_, c2_];

let st_transitions =
   not (privileged c2) ==>
      c1:-"g" <- c1:-"r"
|||not (privileged c1) ==>
      c2:-"g" <- c2:-"r"
|||TRUE ==>
      c1:-"r" <- not (c1:-"g")
|||TRUE ==>
      c1:-"d" <- c1:-"g"
|||TRUE ==>
      c2:-"r" <- not (c2:-"g")
|||TRUE ==>
      c2:-"d" <- c2:-"g";
```