Reactive Modules

Rajeev Alur*
Bell Laboratories

Thomas A. Henzinger†
Electrical Engineering and Computer Sciences
University of California at Berkeley

Abstract. We present a formal model for concurrent systems. The model represents synchronous and asynchronous components in a uniform framework that supports compositional (assume-guarantee) and hierarchical (stepwise refinement) reasoning. While synchronous models are based on a notion of atomic computation step, and asynchronous models remove that notion by introducing stuttering, our model is based on a flexible notion of what constitutes a computation step: by applying an abstraction operator to a system, arbitrarily many consecutive steps can be collapsed into a single step. The abstraction operator, which may turn an asynchronous system into a synchronous one, allows us to describe systems at various levels of temporal detail. For describing systems at various levels of spatial detail, we use a hiding operator that may turn a synchronous system into an asynchronous one. We illustrate the model with diverse examples from synchronous circuits, asynchronous shared-memory programs, and synchronous message passing.

1 Introduction

We introduce a new formal model for reactive computation. Our target application is hardware-software codesign and verification. This application requires (1) an ability to describe and compose modules with different synchrony assumptions, (2) an ability to describe and compose modules at different levels of abstraction, and (3) an ability to decompose verification tasks into subtasks of lower complexity. Our model formalizes heterogeneous systems that are built from synchronous and asynchronous hardware and software components, and provides assume-guarantee and abstraction principles for reasoning about such systems. The salient features of our model are *scalability* along both the space and time axes, and *interdefinability* of synchronous and asynchronous behavior.

Scalability. Scalability along the space axis means that spatial implementation details of a module, such as internal variables and wires, can be hidden from outside observers. Scalability along the time axis means that temporal implemementation details, such as internal computation steps and delays, can be hidden from outside observers.

Example. A 64-bit adder can be implemented either by using two 32-bit adders in parallel, or by using a single 32bit adder twice. The first implementation decomposes the 64-bit adder spatially, by splitting it into two components; the second implementation decomposes the 64-bit adder temporally, by splitting each computation step into two micro-steps. Both implementations are presented at the end of Section 5. More generally, spatial scaling provides components at different levels of detail, such as gates, ALUs, and processors; and temporal scaling provides computation steps at different levels of detail, such as gate operations, arithmetic operations, and processor instructions. □

While spatial scalability is a standard feature of concurrency models, the concept of temporal scalability is inspired by the notion of multiform time in synchronous programming languages [7]. In verification, temporal scaling is usually performed in an informal, manual manner under the umbrella buzzword of "abstraction." We introduce temporal scaling as a modeling primitive, called **next**, that supports the formal construction and the automatic analysis of temporal abstractions. If P is a module, and x is an output variable of P, then the more abstract module $Q = \textbf{next}\ x\ \textbf{for}\ P$ combines as many computation steps of P into a single computation step of Q as are required to change the output x. For example, if P is a gate-level description of a processor, and the toggling of x signals the completion of an instruction, then Q

*alur@bell-labs.com; http://cm.bell-labs.com/who/alur.
†tah@eecs.berkeley.edu; http://www.eecs.berkeley.edu/~tah.
Supported in part by the ONR YIP award N00014-95-1-0520, by the NSF CAREER award CCR-9501708, by the NSF grant CCR-9504469, by the AFOSR contract F49620-93-1-0056, by the ARPA grant NAG2-892, and by the SRC contract 95-DC-324A.

is an instruction-level description of the processor.

Interdefinability. In fully synchronous behavior, concurrent modules proceed in lock-step and respond to mutual inputs by simultaneous outputs. In fully asynchronous behavior, concurrent modules proceed by interleaving and respond to inputs by eventual outputs. Interdefinability means that after hiding spatial information, a collection of synchronous modules can appear asynchronous to outside observers; and after hiding temporal information, a collection of asynchronous modules can appear synchronous.

Example. Consider a transducer that accepts integers as input and computes corresponding squares as output. At an abstract level, the transducer may proceed synchronously, in discrete rounds, accepting one integer per round and computing one square per round; or it may proceed asynchronously, accepting a stream of integers and computing an arbitrarily delayed stream of squares. The (a)synchrony of the abstract transducer, however, is independent of whether a concrete implementation of the transducer employs synchronous modules or asynchronous modules or both. For example, a distributed asynchronous transducer can be implemented using synchronous communication on hidden channels; and a synchronous transducer can be implemented using delay-insensitive circuitry whose internal computation steps and delays are hidden. □

Overview. The paper defines the formalism of reactive modules. Definitions are usually preceded by motivating thoughts, and succeeded by illustrative examples as well as properties that ensure the soundness of the definitions. The bulk of the paper discusses safety aspects of reactive modules. Fair reactive modules are defined in Section 6, and their properties will be discussed in a more comprehensive version of this paper.

2 Definition of Reactive Modules

Variables vs. events. A (reactive) module P has a finite set of typed variables, denoted X_P. A *state* of P is a valuation for the variables in X_P. Events, such as clock ticks, are modeled by toggling boolean variables. The event that is represented by the boolean variable *tick* occurs whenever the module proceeds from a state s to a state t such that $s[tick] \neq t[tick]$.

System vs. environment. The module P represents a system that interacts with an environment. Some of the variables in X_P are updated by the system, and the other variables in X_P are updated by the environment. Hence, the set X_P is partitioned into two sets: the set $ctrX_P$ of *controlled variables*, and the set $extlX_P$ of *external variables*.

$privX_P$	$intfX_P$	$extlX_P$
$ctrX_P$		
	$obsX_P$	
X_P		

Table 1: Module variables

States vs. observations. Not all controlled variables of the module P are visible to the environment. Hence, the set $ctrX_P$ is partitioned further into two sets: the set $privX_P$ of *private variables*, and the set $intfX_P$ of *interface variables*. The interface variables and the external variables are *observable*, denoted $obsX_P$. An *observation* of P is a valuation for the variables in $obsX_P$. The various classes of variables and their relationships are summarized in Table 1.

Asynchrony vs. synchrony. During the execution of the module P the variables in X_P change their values in a sequence of rounds. Various models of reactivity propose different ways in which the variables are updated in a single round.

Pure asynchrony (interleaving [5, 9, 12, 14]): either the system updates the controlled variables, or the environment updates the external variables.

Observable asynchrony (I/O automata [13]): either the system updates the controlled variables, or the environment updates the external variables and the system updates the private variables in response.

Atomic synchrony (Mealy machines [10, 15]; CSP rendezvous [8, 16]): the system and the environment simultaneously update variables in an interdependent fashion.

Nonatomic synchrony (synchronous programming languages [3, 4, 7]): each round (macro-step) consists of several subrounds (micro-steps), and the system and the environment take turns in executing micro-steps to update variables.

The first two options lead to nonblocking communication, provided the system is always prepared to respond to all possible environment moves. Nonblocking communication supports compositional reasoning

with respect to a trace semantics. The third option leads to the possibility of deadlocks, and the fourth option may lead to the possibility of nonterminating computation within a round.

We use the power of nonatomic synchrony, but restrict it to ensure nonblocking communication. First, each variable is updated in exactly one subround of each round. Second, the controlled variables of a module are partitioned into groups called atoms, and the variables within a group are updated simultaneously, in the same subround of each round. Third, the atoms are partially ordered. If atom A precedes atom B in the partial ordering, then in each round, the A-subround must precede the B-subround, and the updated values of the variables controlled by B may depend on the updated values of the variables controlled by A.

Latched vs. updated values. In each round, every variable x has two values. The value of x at the beginning of the round is called the *latched value*, and the value of x at the end of the round is called the *updated value*. We use unprimed symbols, such as x, to refer to latched values, and primed symbols, such as x', to refer to the corresponding updated values.

Initial vs. update actions. The module P proceeds in a sequence of rounds. The first round is an *initialization round*, during which the variables in X_P are initialized. Each subsequent round is an *update round*, during which the variables in X_P are updated. The initialization and updating of variables are specified by actions. An *action* from X to Y is a binary relation between the valuations for X and the valuations for Y. The action α from X to Y is *executable* if for every valuation s for X, the number of valuations t for Y with $(s,t) \in \alpha$ is nonzero and finite. Executable actions are enabled in all states and ensure finitely branching nondeterminism.

Definition 2.1 [**Atom**] *Let X be a finite set of typed variables. An X-*atom *A consists of a declaration and a body. The atom declaration consists of a set $ctrX_A \subseteq X$ of* controlled variables, *a set $readX_A \subseteq X$ of* read variables, *and a set $waitX_A \subseteq X \backslash ctrX_A$ of* awaited variables. *The atom body consists of an executable* initial action *$Init_A$ from $waitX'_A$ to $ctrX'_A$ and an executable* update action *$Update_A$ from $readX_A \cup waitX'_A$ to $ctrX'_A$.*

The atom A initializes and updates the variables in $ctrX_A$. In the initial round, the initial action assigns initial values to the variables in $ctrX_A$ as a nondeterministic function of the initial values of the awaited variables. In each update round, the update action assigns updated values to the variables in $ctrX_A$ as a nondeterministic function of the latched values of the read variables and the updated values of the awaited variables. Hence, in each round, the A-subround can take place only after all variables in $waitX_A$ have been updated. The variable x *awaits* the variable y, written $x \succ_A y$, if $x \in ctrX_A$ and $y \in waitX_A$.

Definition 2.2 [**Module**] *A* (reactive) module *P consists of a declaration and a body. The module declaration is a finite set X_P of typed variables that is partitioned as shown in Table 1. The module body is a set $\mathcal{A}_P$ of X_P-atoms such that (1) $(\cup_{A \in \mathcal{A}_P} ctrX_A) = ctrX_P$; (2) for all atoms A and B in $\mathcal{A}_P$, $ctrX_A \cap ctrX_B = \emptyset$; and (3) the transitive closure $\succ_P = (\cup_{A \in \mathcal{A}_P} \succ_A)^+$ is asymmetric.*

The first two conditions ensure that the atoms of P control precisely the variables in $ctrX_P$, and that each variable in $ctrX_P$ is controlled by precisely one atom. The third condition ensures that the await dependencies among the variables in X_P are acyclic. A linear ordering $A_0, \ldots, A_k$ of the atoms in $\mathcal{A}_P$ is *consistent* if for all $0 \leq i < j \leq k$, the awaited variables of A_i are disjoint from the controlled variables of A_j. The asymmetry of $\succ_P$ ensures that there exists a consistent ordering.

Module execution. When executing the module P, in each round, first the external variables are assigned arbitrary values, and then the atoms in $\mathcal{A}_P$ are executed in an arbitrary consistent order. In this manner, every round can be completed, and all nondeterminism in the completion of a round is caused by nondeterminism of the environment and of individual atoms, not by the ordering of the atoms.

3 Examples of Reactive Modules

While lack of space does not permit us to give a formal definition of our syntax for specifying atoms and modules, our examples will be comprehensible once a few conventions are explained. Variable declarations are indicated by keywords such as **awaits**, for the awaited variables of an atom, and **private**, for the private variables of a module. Initial and update actions are specified by the keywords **init** and **update**, followed by guarded commands. If several guards are true, then one of the corresponding assignments is chosen nondeterministically; if none of the

```
module And
  interface out: 𝔹
  external in1, in2: 𝔹
  atom out awaits in1', in2'
    init update
      [] in1' = 0 ∧ in2' = 0 → out' := 0
      [] in1' = 0 ∨ in2' = 0 → out' := 0
      [] in1' = 1 ∧ in2' = 1 → out' := 1

module Latch
  private state : 𝔹
  interface out: 𝔹
  external set, reset: 𝔹
  atom out reads state
    update
      [] true → out' := state
  atom state awaits set', reset'
    init update
      [] set' = 1 → state' := 1
      [] reset' = 1 → state' := 0
```

Figure 1: Synchronous AND gate and latch

guards are true, then all controlled variables stay unchanged. The combined keyword **init update** indicates that the following guarded command specifies both the initial and update actions. We omit initial commands if the controlled variables have finite types and may be initialized arbitrarily.

Synchronous circuits. The module *And* of Figure 1 models a synchronous AND gate that produces the boolean output *out* as a function of the two boolean inputs in_1 and in_2. In each round, the interface variable *out* is updated after both external variables in_1 and in_2 have been updated. The synchronous latch *Latch* takes the two boolean inputs *set* and *reset*, produces the boolean output *out*, and maintains the private bit *state*. In each update round, the latch copies its state to the interface variable *out*, without waiting for the updated values of the external variables. In a later subround, after both external variables have been updated, the latch updates its state: if both updated external variables are low, then *state* stays unchanged; if only *set* is high, then *state* goes to 1; if only *reset* is high, then *state* goes to 0; if both are high, then *state* goes to an arbitrary value.

Remark on history-free variables. Given a module P, a variable x of P is *history-free* if x is not read by any atom of P. Syntactically, the variable x is history-free iff the update commands of P reference only the updated value x' and not the latched value x. In synchronous circuits, all variables that represent wires are history-free; for example, all variables of Figure 1 except the latch state are history-free. In each round, the possible updated values of a history-free variable depend only on the latched values of variables that are not history-free, and on the updated values of other variables. Hence, during the verification of a module, the values of history-free variables can be omitted from the search stack. □

```
module P1
  interface pc1: {outCS, reqCS, inCS}; x1: 𝔹
  external pc2: {outCS, reqCS, inCS}; x2: 𝔹
  atom pc1, x1 reads pc1, pc2, x1, x2
    init
      [] true → pc1' := outCS
    update
      [] pc1 = outCS → pc1' := reqCS; x1' := x2
      [] pc1 = reqCS ∧ (pc2 = outCS ∨ x1 ≠ x2) →
           pc1' := inCS
      [] pc1 = inCS → pc1' := outCS
      [] true →

module P2
  interface pc2: {outCS, reqCS, inCS}; x2: 𝔹
  external pc1: {outCS, reqCS, inCS}; x1: 𝔹
  atom pc2, x2 reads pc1, pc2, x1, x2
    init
      [] true → pc2' := outCS
    update
      [] pc2 = outCS → pc2' := reqCS; x2' := ¬x1
      [] pc2 = reqCS ∧ (pc1 = outCS ∨ x1 = x2) →
           pc2' := inCS
      [] pc2 = inCS → pc2' := outCS
      [] true →
```

Figure 2: Asynchronous mutual-exclusion protocol

Asynchronous shared-memory programs. The modules P_1 and P_2 of Figure 2 model the two processes of Peterson's solution to the mutual-exclusion problem for shared variables. Each process P_i has a program counter pc_i and a flag x_i, both of which can be observed by the other process. The program counter indicates whether a process is outside its critical section ($pc_i = outCS$), requesting the critical section ($pc_i = reqCS$), or occupying the critical section ($pc_i = inCS$). In each update round, a process looks at the latched values of all variables and, nondeterministically, either updates its controlled variables or sleeps (i.e., leaves the controlled variables unchanged), without waiting to see what the other process does.

Remark on interleaving. Unlike in interleaving mod-

els, both processes may modify their variables in the same round. While the protocol ensures mutual exclusion even under these weaker conditions, if one were to insist on the interleaving assumption, one would add a third module that, in each update round, nondeterministically schedules either or none of the two processes. The modeling of interleaving by a scheduler module introduces only history-free variables, and thus, does not increase the search space during verification. □

Remark on write-shared variables. The original formulation of Peterson's protocol uses a single write-shared boolean variable x, whose value always corresponds to the value of the predicate $x_1 = x_2$ in our formulation. If one were to insist on modeling x as a write-shared variable, one would add a third module with the interface variable x, and with two external variables that are used by the two processes for modifying x. Thus, a variable that is write-shared among n processes is modeled by $(n+1)$ separate variables. Since n of these variables are history-free, the modeling does not increase the search space during verification. □

Synchronous message passing. The modules *Sender* and *Receiver* of Figure 3 communicate via events in order to transmit a stream of messages. We write $x : \mathbb{E}$ to declare x to be a boolean variable that is used for modeling events. To issue an event represented by x, we write $x!$, which stands for the assignment $x' := \neg x$. To check if an event represented by x is present, we write $x?$, which stands for the predicate $x' \neq x$.

The private variable *pc* of the sender indicates if it is producing a message ($pc = produce$), or attempting to send a message ($pc = send$). The private variable *pc* of the receiver indicates if it is waiting to receive a message ($pc = receive$), or consuming a message ($pc = consume$). Messages are produced by the atom *A*Prod, which requires an unknown number of rounds to produce a message. Once a message is produced, the event $done_P$ is issued, and the message is shown as msg_P (the actual value of message is chosen nondeterministically from the finite type $\mathbb{M}$). Once a message has been produced, the sender is ready to send the message, and *pc* is updated. When ready to send a message, the sender sleeps until the receiver becomes ready to receive, and when ready to receive a message, the receiver sleeps until the sender transmits a message.

The synchronization of both agents is achieved by two-way handshaking in three subrounds within a single update round. The first subround belongs to the

module *Sender*
  private pc: $\{produce, send\}$
  interface $transmit, done_P : \mathbb{E}$; $msg_S, msg_P : \mathbb{M}$
  external $ready : \mathbb{E}$
  atom $pc, transmit, msg_S$
    reads $pc, done_P, msg_P, ready$
    awaits $done'_P, ready'$
    init
    [] $true \rightarrow pc' := produce$
    update
    [] $pc = produce \wedge done_P? \rightarrow pc' := send$
    [] $pc = send \wedge ready? \rightarrow$
      $transmit!$; $msg'_S := msg_P$; $pc' := produce$
  *A*Prod : **atom** $done_P, msg_P$ **reads** $pc, done_P$
    update
    [] $pc = produce \rightarrow done_P!$; $msg'_P := \mathbb{M}$
    [] $true \rightarrow$

module *Receiver*
  private pc: $\{receive, consume\}$; $msg_R : \mathbb{M}$
  interface $ready, done_C : \mathbb{E}$; $msg_C : \mathbb{M}$
  external $transmit : \mathbb{E}$; $msg_S : \mathbb{M}$
  atom pc, msg_R
    reads $pc, transmit, done_C$
    awaits $transmit', msg'_S, done'_C$
    init
    [] $true \rightarrow pc' := receive$
    update
    [] $pc = receive \wedge transmit? \rightarrow$
      $msg'_R := msg'_S$; $pc' := consume$
    [] $pc = consume \wedge done_C? \rightarrow pc' := receive$
  atom $ready$ **reads** pc
    update
    [] $pc = receive \rightarrow ready!$
    [] $true \rightarrow$
  *A*Cons : **atom** $done_C, msg_C$ **reads** $pc, done_C, msg_R$
    update
    [] $pc = consume \rightarrow done_C!$; $msg'_C := msg_R$
    [] $true \rightarrow$

Figure 3: Synchronous message-passing protocol

receiver. If the receiver is ready to receive a message, it issues the interface event *ready* to signal its readiness to the sender. The second subround belongs to the sender. If the sender sees the external event *ready* and is ready to send a message, it issues the interface event *transmit* to signal a transmission. The third subround belongs to the receiver. If the receiver sees the external event *transmit*, it copies the message from the external variable msg_S to the private variable msg_R. The sender goes on to wait for the production of another message, and the receiver goes on to consume msg_R. Messages are consumed by the atom ACons, which requires an unknown number of rounds to consume a message. Once a message is consumed, the event $done_C$ is issued, the consumed message is shown as msg_C, and the receiver waits to receive another message.

Remark on event variables. While not history-free, boolean variables that represent events can also be omitted from the search stack during verification. This is because the actual value of an event variable is immaterial. □

Combinational, lazy, and event-driven atoms. We identify three special classes of atoms. An atom is *combinational* if it has no read variables. We say that an atom *sleeps* in an update round if the values of all atom variables stay unchanged. An X-atom A is *lazy* if it may sleep in every update round: for all valuations s and t for X, $(s[readX_A] \cup t'[waitX'_A], s'[ctrX'_A]) \in Update_A$.[1] A sufficient syntactic condition for laziness is the presence of the option $[\!]\ true \rightarrow$ in the update command. The X-atom A is *event-driven* if it sleeps in every update round in which no external variable that is both read and awaited changes its value (in particular, no external event is present): for all valuations s for X, $(s[readX_A] \cup s'[waitX'_A], s'[ctrX'_A]) \in Update_A$. A sufficient syntactic condition for being event-driven is the presence of a conjunct of the form $x?$ in each guard of the update command. Also, every lazy atom is event-driven. For example, the atom of the module *And* from Figure 1 is combinational, the atom of the module P_1 from Figure 2 is lazy, and the modules *Sender* and *Receiver* from Figure 3 have event-driven atoms.

[1] Given a valuation s for the set X of variables, and a subset Y of X, we write $s[Y]$ for the projection of s to the variables in Y. If s is a valuation for a set X of unprimed variables, then s' denotes the valuation for the set X' of corresponding primed variables such that s' assigns to each variable x' the value $s[x]$. Given two disjoint sets X and Y of variables, if s is a valuation for X and t is a valuation for Y, then $s \cup t$ denotes the combined valuation for $X \cup Y$.

Definition 3.1 [Asynchrony] *A module is* asynchronous *if all interface variables are controlled by lazy atoms.*

We say that a module *stutters* in an update round if the values of all interface variables stay unchanged. Then an asynchronous module may stutter in every update round. It follows that the environment cannot enforce observable progress of an asynchronous module. While an asynchronous module can privately record all updates of external variables, all updates of interface variables proceed at a speed that is independent of the environment speed. For example, the module P_1 from Figure 2 is asynchronous.

4 Semantics of Reactive Modules

Trace language. A state s of the module P is *initial* if it can be obtained by executing all initial actions of P in a consistent order: for each atom $A \in \mathcal{A}_P$, $(s'[waitX'_A], s'[ctrX'_A]) \in Init_A$. For two states s and t of P, the state t is a *successor* of s if t can be obtained from s by executing all update actions of P in a consistent order: for each atom $A \in \mathcal{A}_P$, $(s[readX_A] \cup t'[waitX'_A], t'[ctrX'_A]) \in Update_A$. A *trajectory* of P is a finite sequence $s_0 \dots s_n$ of states such that (1) the first state s_0 is initial and (2) for all $0 \leq i < n$, the state s_{i+1} is a successor of s_i. If $\overline{s} = s_0 \dots s_n$ is a trajectory of P, then the corresponding sequence $\overline{s}[obsX_P] = s_0[obsX_P] \dots s_n[obsX_P]$ of observations is a *trace* of P. The *trace language* of P, denoted L_P, is the set of traces of P.

Proposition 4.1 *For every module P, the trace language L_P is prefix-closed and contains traces of arbitrary length.*

Implementation preorder. The semantics of the module P consists of the trace language L_P, as well as all information that is necessary for describing the possible interactions of P with the environment: the set $intfX_P$ of interface variables, the set $extlX_P$ of external variables, and the await dependencies $\succ_P \cap (intfX_P \times obsX_P)$ between interface variables and observable variables.

Definition 4.1 [Implementation] *The module P* implements *the module Q, written $P \preceq Q$, if the following conditions are met: (1) every interface variable of Q is an interface variable of P; (2) every external variable of Q is an observable variable of P; (3) for all variables x in $obsX_Q$ and all variables y in $intfX_Q$,*

```
module RoundCount
  interface count: ℕ
  atom count reads count
    update
      [] true → count' := count + 1

module SyncCount
  interface count: ℕ
  external tick: 𝔼
  atom count reads count, tick awaits tick'
    update
      [] tick? → count' := count + 1

module AsyncCount
  interface count: ℕ
  atom count reads count
    update
      [] true → count' := count + 1
      [] true →
```

Figure 4: Three counters

if $y \succ_Q x$, then $y \succ_P x$; and (4) if $\bar{s}$ is a trace of P, then the projection $\bar{s}[obsX_Q]$ is a trace of Q.

The first three conditions ensure that the compatibility constraints imposed by P on its environment are stronger than those imposed by Q. The fourth condition is conventional trace inclusion.

Proposition 4.2 *The implementation relation $\preceq$ is a preorder (i.e., reflexive and transitive).*

We write $P \cong Q$ if P implements Q and Q implements P. It follows that $\cong$ is an equivalence relation.

Round-insensitivity. Round-insensitive modules are modules with trace languages that are closed under the insertion of stutter steps. A module is *round-insensitive* if it has only combinational and event-driven atoms. For example, the modules *Sender* and *Receiver* from Figure 3 are round-insensitive. We say that a module *sleeps* in an update round if the values of all controlled variables stay unchanged, and the environment *stutters* in an update round if the values of all external variables stay unchanged. A round-insensitive module may sleep in every update round in which the environment stutters. For round-insensitive modules P and Q, it follows that $L_P \subseteq L_Q$ iff the stutter closure of L_P is a subset of the stutter closure of L_Q.

The difference between asynchrony and round-insensitivity is illustrated by the three counters shown in Figure 4. While the environment cannot enforce observable progress of an asynchronous module, it can enforce observable progress of a round-insensitive module by modifying external variables. A round-insensitive module, on the other hand, cannot count rounds, but only changes in the values of external variables. In our example, the round-sensitive synchronous counter *RoundCount* is incremented in each round, the round-insensitive synchronous counter *SyncCount* is incremented with every occurrence of the external event *tick*, and the round-insensitive asynchronous counter *AsyncCount* is incremented nondeterministically.

5 Operations on Reactive Modules

We create new modules using variable renaming, parallel composition, variable hiding, round abstraction, and triggering.

Variable renaming. The renaming operation is useful for creating different instances of a module, and for avoiding name conflicts. Let P be a module, and let x and y be two variables of the same type such that y is not in X_P. Then the module $P[x := y]$ results from P by renaming x to y. For any two modules, we henceforth assume that a private variable of one module is not a module variable of the other module; this can always be achieved by renaming private variables.

Parallel composition. The two modules P and Q are *compatible* if (1) the interface variables of P and Q are disjoint, and (2) the await dependencies among observable variables of P and Q are acyclic —that is, the transitive closure $(\succ_P \cup \succ_Q)^+$ is asymmetric. It follows that if P and R are compatible modules, and $P \preceq Q$, then Q and R are also compatible.

Definition 5.1 [Composition] *If P and Q are two compatible modules, then the* composition *$P||Q$ is the module with the set $privX_{P||Q} = privX_P \cup privX_Q$ of private variables, the set $intfX_{P||Q} = intfX_P \cup intfX_Q$ of interface variables, the set $extlX_{P||Q} = (extlX_P \cup extlX_Q) \backslash intfX_{P||Q}$ of external variables, and the set $\mathcal{A}_{P||Q} = \mathcal{A}_P \cup \mathcal{A}_Q$ of atoms.*

It is easy to check that for two compatible modules P and Q, the composition $P||Q$ is again a module. The composition $P||Q$ is asynchronous iff both P and Q are asynchronous, and $P||Q$ is round-insensitive iff both P and Q are round-insensitive. Henceforth, whenever we write $P||Q$, we assume that P and Q are compatible.

Proposition 5.1 *The composition operator has the following properties.*

(1) $\|$ is commutative and associative.
(2) $L_{P\|Q} = L_P \cap L_Q$.
(3) $P\|Q \preceq P$.
(4) If $P \preceq Q$, then $P\|R \preceq Q\|R$.
(5) If $P\|Q' \preceq P'$ and $P'\|Q \preceq Q'$, then $P\|Q \preceq P'\|Q'$.

Thus, parallel composition behaves like logical conjunction. Property (4) asserts that $\preceq$ is a congruence with respect to composition, and property (5) asserts an assume-guarantee principle for verification [1, 2, 6]. In order to prove that a complex module $P_1\|P_2$ (with a large state space) implements a simple specification Q (with a small state space), by (4) it suffices to find two modules Q_1 and Q_2 such that (a) P_1 implements Q_1, (b) P_2 implements Q_2, and (c) $Q_1\|Q_2$ implements Q. If this is not possible, by (5) it suffices to replace the proof obligations (a) and (b) by the weaker proof obligations (a') $P_1\|Q_2$ implements Q_1 and (b') $Q_1\|P_2$ implements Q_2, which still avoid the construction of the complex product $P_1\|P_2$. The validity of (5) crucially depends on the executability of initial and update actions.

Variable hiding. After composing two modules, it may be appropriate to convert some interface variables to private variables, so that they are used only for the interaction of the component modules, and are no longer visible to the environment of the composite module. Let P be a module, and let x be a variable. If x is an interface variable of P, then the module **hide** x **in** P is obtained from P by moving x from the interface variables to the private variables. If x is not an interface of P, then **hide** x **in** P is identical to P.

Proposition 5.2 *The hiding operator has the following properties.*

(1) **hide** x **in hide** y **in** P = **hide** y **in hide** x **in** P.
(2) $P \preceq$ **hide** x **in** P.
(3) If $P \preceq Q$, then **hide** x **in** $P \preceq$ **hide** x **in** Q.

Hiding preserves both asynchrony and round-insensitivity, but not synchrony. Hence, hiding is useful for constructing asynchronous modules from synchronous modules. Consider, for example, an asynchronous module *Clock* that nondeterministically issues the interface event *tick*:

module *Clock*
  interface *tick*: $\mathbb{E}$
  atom *tick* **reads** *tick*
    update
      [] *true* $\rightarrow$ *tick*!
      [] *true* $\rightarrow$

Then, given the synchronous counter *SyncCount* from Figure 4, we can implement the asynchronous counter *AsyncCount* using hiding:

$$AsyncCount \cong \textbf{hide } tick \textbf{ in } SyncCount \parallel Clock$$

By composing the synchronous modules *Sender* and *Receiver* from Figure 3, we obtain a synchronous message passing protocol. After hiding the communication events, so that only the streams of produced and consumed messages remain visible, we obtain the asynchronous module

$$SendRec = \textbf{hide } ready, transmit, msg_S \textbf{ in } Sender \parallel Receiver$$

which can be shown $\cong$-equivalent to a message-passing protocol that uses asynchronous, rather than synchronous, handshaking.

Round abstraction. In order to reduce the complexity of a system, it is often useful to combine several consecutive rounds into a single, more abstract round. For this purpose, we introduce the abstraction operator **next**. Intuitively, given a subset Y of the interface variables of the module P, the module **next** Y **for** P collapses consecutive rounds of P until one of the variables in Y changes its value.

A *controlled state* of the module P is a valuation for the controlled variables of P, and an *external state* of P is a valuation for the external variables of P. For two external states t and u of P, an *iteration of P from t to u* is a finite sequence $\overline{s} = s_0 \dots s_n$ of controlled states of P such that $n \geq 1$ and for all $0 \leq i < n$, the state $s_{i+1} \cup u$ is a successor of the state $s_i \cup t$. Hence, along an iteration, the update actions of a module are iterated while the latched and the updated values of the external variables stay unchanged. The iteration $\overline{s}$ *modifies* the set $Y \subseteq ctrX_P$ of controlled variables if (1) $s_n[Y] \neq s_0[Y]$ and (2) for all $0 \leq i < n$, $s_i[Y] = s_0[Y]$. If the iteration $\overline{s}$ modifies Y, then the state $s_n \cup u$ is called a *Y-successor* of the state $s_0 \cup t$.

A *round marker* for the module P is a (nonempty) set $Y \subseteq intfX_P$ of interface variables such that for all states s and t of P, there are nonzero and finitely many Y-successors u of s such that u and t agree on the values of all external variables of P. If Y is a round marker for P, then from any state, after the awaited external variables have been updated, the update actions of P can be iterated in a way that leads to the modification of an interface variable in Y. For a finite-state module P, it can be checked automatically if Y is a round marker for P, by checking an $\exists\mathcal{U}$-formula of branching-time logic.

Definition 5.2 [Abstraction] *If Y is a round marker for the module P, then the abstraction* **next** Y **for** P *is a module with the same declaration as P and a single atom, A_P^Y. The atom A_P^Y has the set $ctrX_P$ of controlled variables, the set $readX_P = (\cup_{A \in \mathcal{A}_P} readX_A)$ of read variables, and the set $waitX_P = (\cup_{A \in \mathcal{A}_P} waitX_A)$ of awaited variables. The initial action of A_P^Y contains all pairs of the form $(s'[waitX'_P], s'[ctrX'_P])$, where s is an initial state of P. The update action of A_P^Y contains all pairs of the form $(s[readX'_P] \cup t'[waitX'_P], t'[ctrX'_P])$, where t is a Y-successor of s.*

It is easy to check that if Y is a round marker for P, then the abstraction **next** Y **for** P is again a module. Henceforth, whenever we write **next** Y **for** P, we assume that Y is a round marker for P. The module **next** $intfX_P$ **for** P is called the *stutter reduction* of P, and denoted **next** P. In each update round, the stutter reduction **next** P iterates the update actions of P until some interface variable changes. The **next** operator supports compositional reasoning, because $\preceq$ is a congruence with respect to abstraction.

Proposition 5.3 *If $P \preceq Q$, then* **next** Y **for** $P \preceq$ **next** Y **for** Q.

Abstraction is useful for constructing synchronous modules from asynchronous modules. For example, given the asynchronous counter *AsyncCount* from Figure 4, we can implement the synchronous counter *RoundCount* using abstraction:

$$RoundCount = \textbf{next}\ AsyncCount$$

Similarly, while the module *SendRec* is asynchronous, its stutter reduction

$$RedSendRec = \textbf{next}\ SendRec$$

is synchronous. In each round of *RedSendRec*, either a message is produced by the atom AProd or a message is consumed by the atom ACons, or both.

Remark on verification. Temporal properties and implementation relations for finite-state modules can be checked algorithmically, by constructing the state-transition graph G_P that underlies a module P. Consider the abstraction $Q =$ **next** Y **for** P. The search of G_Q may be more efficient than the search of G_P, because abstraction may cause some variables to become history-free. More importantly, G_Q typically has many fewer edges than G_P, and therefore a smaller reachable state space. When G_Q is searched on the fly, the reachable states of P never have to be added to the search stack. Rather, the edges of G_Q are constructed by a secondary search in G_P, which is implemented using an auxiliary stack that is released once all edges from a given vertex of G_Q have been found. This reduction of the reachable state space is similar to synchronous programming languages, where only macro-steps, rather than micro-steps, correspond to edges in the state-transition graph [7]. For example, in every reachable state of the module *RedSendRec*, either *pc* of the sender equals *send* or *pc* of the receiver equals *receive* (this invariant does not hold for the module *SendRec*). □

Triggering. When implementing an environment P for the abstraction **next** Y **for** R at the round granularity of R, we must ensure that P does not change its interface during rounds of R that are collapsed. For this purpose, we introduce the operator **trigger** as a dual of **next**. Intuitively, for a subset Y of the external variables of the module P, the module **trigger** Y **for** P sleeps until some external variable in Y changes its value. Then, P is executed.

A *trigger* for the module P is a set $Y \subseteq extlX_P$ of external variables.

Definition 5.3 [Trigger] *If Y is a trigger for the module P, then the module $Q =$* **trigger** Y **for** P *has the same declaration as P and a single atom, B_P^Y. The atom B_P^Y has the set $ctrX_P$ of controlled variables, the set $readX_P \cup Y$ of read variables, and the set $waitX_P \cup Y$ of awaited variables. The initial action of B_P^Y contains all pairs of the form $(s'[waitX'_Q], s'[ctrX'_Q])$, where s is an initial state of P. The update action of B_P^Y contains all pairs of the form $(s[readX_Q] \cup t'[waitX'_Q], t'[ctrX'_Q])$, where either (1) $s[Y] = t[Y]$ and $s[ctrX_P] = t[ctrX_P]$, or (2) $s[Y] \neq t[Y]$ and t is a successor of s.*

Henceforth, whenever we write **trigger** Y **for** P, we assume that Y is a set of variables that contains no controlled variables of P. If Y contains some variables that are not (external) variables of P, then we agree that **trigger** Y **for** P stands for the module **trigger** Y **for** $(P||Q)$, where Q is the trivial module with the external variables $Y \backslash extlX_P$ and no atoms. The round-insensitive module **trigger** $extlX_P$ **for** P is called the *event reduction* of P, and denoted **trigger** P. In each update round, the event reduction **trigger** P executes P in those update rounds in which the value of some external variable changes, and otherwise sleeps.

Proposition 5.4 *The trigger operator has the following properties.*

(1) **trigger** Y **for trigger** Z **for** $P \cong$
trigger $Y \cap Z$ **for** P.
(2) If $P \preceq Q$, then
trigger Y **for** $P \preceq$ **trigger** Y **for** Q.
(3) If Y contains all external read variables of Q, then
next Y **for** $(P \parallel (\textbf{trigger}\ Y\ \textbf{for}\ Q)) \cong$
next $((\textbf{next}\ Y\ \textbf{for}\ P) \parallel Q)$.

Property (2) asserts that $\preceq$ is a congruence with respect to triggering. Property (3) asserts that **trigger** is the dual of **next**. The outer abstraction operator of the right-hand side serves the sole purpose of compressing the atoms of the module $(\textbf{next}\ Y\ \textbf{for}\ P) \parallel Q$ into a single atom. Hence, we obtain the following abstraction principle for verification: in order to prove that a Y-abstraction of a module of the form $P \parallel (\textbf{trigger}\ Y\ \textbf{for}\ Q)$ implements the specification R, it suffices to find two modules R_1 and R_2 such that (a) the Y-abstraction of P implements R_1, (b) Q implements R_2, and (c) $R_1 \| R_2$ implements R.

Using triggering, we can implement the round-insensitive counter *SyncCount* of Figure 4 from the round-sensitive counter *RoundCount*:

$$\mathit{SyncCount} \;=\; \textbf{trigger}\ \mathit{tick}\ \textbf{for}\ \mathit{RoundCount}$$

This completes our demonstration that all three counters from Figure 4 are interdefinable.

Example: synchronous circuits. Renaming, composition, and hiding allow us to construct a space hierarchy of modules. For example, using the modules *And* and *Latch* from Figure 1, and a module *Not* that models a synchronous inverter, we can construct synchronous circuits. For instance, the circuit

$$\begin{array}{l} \mathit{Or} = \textbf{hide}\ z_1, z_2, z_3\ \textbf{in} \\ \qquad \parallel \mathit{And}[in_1, in_2, out := z_1, z_2, z_3] \\ \qquad \parallel \mathit{Not}[in, out := in_1, z_1] \\ \qquad \parallel \mathit{Not}[in, out := in_2, z_2] \\ \qquad \parallel \mathit{Not}[in, out := z_3, out] \end{array}$$

is $\cong$-equivalent to a synchronous OR gate with inputs in_1 and in_2 and output out. The notion of a round (clock-cycle) remains unchanged for all modules (circuits) that are constructed in this way. The acyclicity requirement for awaits dependencies prohibits combinational cycles.

The **next** operator changes the notion of a round, and allows us to construct a time hierarchy of modules. Consider the specification *Add64* of a 64-bit (Figure 5). The 32-bit adder *Add32* is specified similarly. We give two implementations of *Add64* using *Add32*. If x is a 64-bit word, we write x_0 for the least significant 32 bits and x_1 for the most significant 32 bits.

$$\begin{array}{l} \textbf{module}\ \mathit{Add64} \\ \quad \textbf{interface}\ z : \mathbb{B}[0..63];\ \mathit{ofl} : \mathbb{B} \\ \quad \textbf{external}\ x, y : \mathbb{B}[0..63];\ \mathit{carry} : \mathbb{B} \\ \quad \textbf{atom}\ z, \mathit{ofl}\ \textbf{awaits}\ x', y', \mathit{carry}' \\ \qquad \textbf{init update} \\ \qquad\quad [\!]\ \mathit{true} \;\rightarrow\; z' := (x' + y' + \mathit{carry}')\ \textbf{mod}\ 2^{64}; \\ \qquad\qquad \mathit{ofl}' := (x' + y' + \mathit{carry}')\ \textbf{div}\ 2^{64} \\ \\ \mathit{ParAdd} \;=\; \textbf{hide}\ u\ \textbf{in} \\ \quad \parallel \mathit{Add32}[x, y, z, \mathit{carry}, \mathit{ofl} := x_0, y_0, z_0, \mathit{carry}, u] \\ \quad \parallel \mathit{Add32}[x, y, z, \mathit{carry}, \mathit{ofl} := x_1, y_1, z_1, u, \mathit{ofl}] \end{array}$$

Figure 5: Parallel implementation of 64-bit adder

The first implementation, *ParAdd* (Figure 5), uses two copies of *Add32* and connects them appropriately. In each update round, *ParAdd* adds two 64-bit words by adding the first half-words and the second half-words in parallel. Hence $\mathit{ParAdd} \preceq \mathit{Add64}$. The second implementation, *SeqAdd* (Figure 6), uses a single copy of *Add32* and embeds it in additional circuitry, represented by the module R. The circuit $\mathit{Add32}[\ldots] \parallel R$ adds the first half-words before adding the second half-words, and requires two consecutive update rounds to compute a 64-bit sum. The completion of the computation is signaled by the event *done*. In *SeqAdd*, the two rounds of each computation are collapsed, so that $\mathit{SeqAdd} \preceq \mathit{Add64}$.

6 Fair Reactive Modules

Based on the trace semantics of modules (Section 4) we can reason about the safety requirements of modules. Reasoning about liveness requirements demands that we consider *infinite* behaviors of modules. Towards this goal, we extend the model of reactive modules by adding fairness constraints, which rule out degenerate infinite behaviors of modules.

Fair modules. An *update choice* of an atom A is a subset of the action Update_A. A *fair atom* is an atom A with a finite set WF_A of weakly fair update choices and a finite set SF_A of strongly fair update choices. Intuitively, a weakly fair update choice cannot be available forever without being taken, and a strongly fair update choice cannot be available infinitely often without being taken. A *fair module* is a reactive module whose atoms are fair atoms.

Fair semantics. An ω-trajectory of a module P is an infinite sequence $\overline{s} = s_0 s_1 \ldots$ of states such that

```
SeqAdd = hide done in next {done} for
  hide a, b, c, u, v, round in
    || Add32[x, y, z, carry, ofl := a, b, c, u, ofl]
    || R

module R
  private v : 𝔹
  interface round : {0, 1}; z : 𝔹[0..63];
      a, b : 𝔹[0..31]; u : 𝔹; done : 𝔼
  external x, y : 𝔹[0..63]; c : 𝔹[0..31]; carry, ofl : 𝔹
  atom round reads round
    init update
      [] round = 0 → round' := 1
      [] round = 1 → round' := 0
  atom a, b, u reads v awaits round', x', y', carry'
    init update
      [] round' = 0 → a' := x'_0; b' := y'_0; u' := carry'
      [] round' = 1 → a' := x'_1; b' := y'_1; u' := v
  atom z, v, done reads done awaits round', c', ofl'
    init
      [] true → z'_0 := c'; v' := ofl'
    update
      [] round' = 0 → z'_0 := c'; v' := ofl'
      [] round' = 1 → z'_1 := c'; done!
```

Figure 6: Sequential implementation of 64-bit adder

(1) the first state s_0 is initial and (2) for all $i \geq 0$, the state s_{i+1} is a successor of s_i. Consider an update choice α of some atom $A \in \mathcal{A}_P$. The update choice α is *enabled* at position i of the ω-trajectory $\overline{s}$ if there is a state t such that $(s_i[readX_A] \cup s'_{i+1}[waitX'_A], t) \in \alpha$. The update choice α is *taken* at position i of $\overline{s}$ if $(s_i[readX_A] \cup s'_{i+1}[waitX'_A], s'_{i+1}[ctrX'_A]) \in \alpha$. The ω-trajectory $\overline{s}$ is *weakly fair* to the update choice α if either α is not enabled at infinitely many positions of $\overline{s}$, or α is taken at infinitely many positions of $\overline{s}$. The ω-trajectory $\overline{s}$ is *strongly fair* to the update choice α if either α is enabled at only finitely many positions of $\overline{s}$, or α is taken at infinitely many positions of $\overline{s}$.

An ω-trajectory $\overline{s}$ of the fair module P is *fair* if for each fair atom $A \in \mathcal{A}_P$, the ω-trajectory $\overline{s}$ is weakly fair to all update choices in WF_A and strongly fair to all update choices in SF_A. If $\overline{s}$ is a fair ω-trajectory of P, then the corresponding infinite sequence $\overline{s}[obsX_P]$ of observations is an *ω-trace* of P. The fair module P *ω-implements* the fair module Q if the first three conditions of Definition 4.1 are met, and (4) if $\overline{s}$ is an ω-trace of P, then the projection $\overline{s}[obsX_Q]$ is an ω-trace of Q.

Remark on receptiveness. Every (finite) trajectory of a fair module P can be extended to a fair ω-trajectory of P (this is the so-called *machine-closure* property). In fact, in order to extend a finite trajectory to a fair ω-trajectory, the module does not need the cooperation of the environment (this is the so-called *receptiveness* property). Assume-guarantee principles for ω-implementation crucially depend on the receptiveness property [2]. □

```
module FP_1
  interface pc_1 : {outCS, reqCS, inCS}; x_1 : 𝔹
  external pc_2 : {outCS, reqCS, inCS}; x_2 : 𝔹
  atom pc_1, x_1 reads pc_1, pc_2, x_1, x_2
    init
      [] true → pc'_1 := outCS
    update weakly-fair α, β
      [] pc_1 = outCS → pc'_1 := reqCS; x'_1 := x_2
      [] pc_1 = reqCS ∧ (pc_2 = outCS ∨ x_1 ≠ x_2) --α-->
           pc'_1 := inCS
      [] pc_1 = inCS --β--> pc'_1 := outCS
      [] true →
```

Figure 7: Fair mutual-exclusion protocol

The operations of renaming, composition, and hiding can be extended to fair modules in the obvious way, and lead to analogues of Propositions 5.1 and 5.2. The relationship between abstraction and fairness is intriguing, and illustrated with the following example.

Example: asynchronous shared-memory programs. Recall Peterson's solution to the mutual-exclusion problem. One process is shown in Figure 7 with fairness constraints; the second fair process, FP_2, is defined similarly. Weak fairness, say, for the update choice β means that along a fair ω-trajectory, it cannot happen that from a certain position onwards, the guard of β is continuously true but the assignment of β is never executed. Thus, the weak fairness of β ensures that the module FP_1 does not remain in its critical section forever. For the fair module $FP_1 \parallel FP_2$ it is possible to prove deadlock freedom: if some process requests to enter the critical section, then eventually some process will be in the critical section (some unfair ω-trajectories do not satisfy this requirement).

Remark on round abstraction vs. fairness. The abstraction operator **next** is closely related to weak fairness. For instance, while not all ω-trajectories of the module $P_1 \parallel P_2$ from Figure 2 satisfy deadlock freedom, all ω-trajectories of the module **next** $(P1 \parallel P_2)$ do. Indeed, the module **next** $(P_1 \parallel P_2)$ satisfies the stronger requirement of *bounded* deadlock freedom: if some process requests to enter the critical section,

then some process will be in the critical section within two rounds. Reasoning about **next**, like reasoning about weak fairness, can be done by iterating safety reasoning: in the case of weak fairness, an ω-trajectory satisfies a property iff infinitely many finite prefixes satisfy a transformed property; in the case of **next**, the safety reasoning is iterated at various levels of round abstraction. In general, however, it is not possible to replace fairness by round abstraction. □

7 Concluding Remarks

We have presented a unified framework for describing reactive computation. In the future, we plan to explore the benefits of this uniformity for computer-aided verification. The efficiency of current verification tools often depends on the specific synchrony assumption supported by the underlying model. For instance, hardware description languages (like VHDL) assume synchronous models, and BDD-based model checking has proved to be successful in this domain. On the other hand, many protocol description languages (like PROMELA [9]) assume asynchronous interleaving, and the most effective verification strategy is enumerative on-the-fly search with reduction techniques based on partial orders and symmetries. Finally, the verification tools for synchronous programming languages (like ESTEREL [3]) can afford to construct global state-transition graphs, because much of the complexity is hidden by the fact that a single transition involves several (sub)transitions between transient states.

While both synchrony and asynchrony can be forced, in one way or another, into most concurrency models, this often comes at the cost of inefficiencies in verification. For example, the use of stutter transitions in synchronous models to represent asynchronous progress increases the number of transitions exponentially over an asynchronous model [11]. Or, the introduction of synchronization points into asynchronous models restricts the applicability of efficient search methods in verification [9]. By contrast, our uniform framework allows us to separate intrinsic truths and complexities about verification methods from accidental and model-dependent idiosyncrasies. Furthermore, our framework supports a variety of proof principles, including assume-guarantee and abstraction rules, which can be used to decompose a verification task into subtasks with smaller state spaces.

Acknowledgements. We thank Bob Kurshan, Ken McMillan, and the VIS group at UC Berkeley for fruitful discussions.

References

[1] M. Abadi and L. Lamport. Conjoining specifications. Technical Report 118, DEC Systems Research Center, Palo Alto, California, 1993.

[2] R. Alur and T.A. Henzinger. Local liveness for compositional modeling of fair reactive systems. In P. Wolper, editor, *CAV 95: Computer-aided Verification*, Lecture Notes in Computer Science 939, pages 166–179. Springer-Verlag, 1995.

[3] G. Berry and G. Gonthier. The synchronous programming language ESTEREL: design, semantics, implementation. Technical Report 842, INRIA, 1988.

[4] G. Berry, S. Ramesh, and R.K. Shyamasundar. Communicating reactive processes. In *Proceedings of the 20th Annual Symposium on Principles of Programming Languages*, pages 85–98. ACM Press, 1993.

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

[6] O. Grümberg and D.E. Long. Model checking and modular verification. *ACM Transactions on Programming Languages and Systems*, 16(3):843–871, 1994.

[7] N. Halbwachs. *Synchronous Programming of Reactive Systems.* Kluwer Academic Publishers, 1993.

[8] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, 1985.

[9] G.J. Holzmann. *Design and Validation of Computer Protocols.* Prentice-Hall, 1991.

[10] R.P. Kurshan. *Computer-aided Verification of Coordinating Processes.* Princeton University Press, 1994.

[11] R.P. Kurshan, M. Merritt, A. Orda, and S.R. Sachs. Modeling asynchrony with a synchronous model. In P. Wolper, editor, *CAV 95: Computer-aided Verification*, Lecture Notes in Computer Science 939, pages 339–352. Springer-Verlag, 1995.

[12] L. Lamport. Specifying concurrent program modules. *ACM Transactions on Programming Languages and Systems*, 5(2):190–222, 1983.

[13] N.A. Lynch. *Distributed Algorithms.* Morgan-Kaufmann, 1996.

[14] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification.* Springer-Verlag, 1992.

[15] K.L. McMillan. *Symbolic model checking: an approach to the state explosion problem.* Kluwer Academic Publishers, 1993.

[16] R. Milner. Calculi for synchrony and asynchrony. *Theoretical Computer Science*, 25(3):267–310, 1983.