Reactive Modules*

Rajeev Alur[†] Thomas A. Henzinger[‡]

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) design and verification. 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 protocols.

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 implementation details, such as internal computation steps and delays, can be hidden from outside observers.

*A preliminary version of this paper appeared in the *Proceedings of the 11th IEEE Symposium on Logic in Computer Science* (LICS), pages 207–218, 1996.

[†]University of Pennsylvania and Bell Laboratories. Address: Department of Computer and Information Science, 200 South 33rd Street, Philadelphia, PA 19104. Email: alur@cis.upenn.edu. URL: http://www.cis.upenn.edu/~alur. Supported in part by the DARPA/NASA grant NAG2-1214.

[‡]University of California at Berkeley. Address: Department of Electrical Engineering and Computer Sciences, Berkeley, CA 94720-17770. Email: tah@eecs.berkeley.edu. URL: 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 DARPA/NASA grant NAG2-1214, and by the SRC contract 97-DC-324.041.

Example. A 64-bit adder can be implemented either by using two 32-bit adders in parallel, or by using a single 32-bit adder twice for each 64-bit addition, first for the lower-order bits and then for the higher-order bits. 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 in 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 [14], and by the notion of action refinement in process algebras [26]. 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 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 the 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 7.

2 Definition of Reactive Modules

A discrete reactive system is a collection of variables that, over time, change their values in a sequence of rounds. We model discrete reactive systems that may interact with each other by mathematical objects that are called *reactive modules* (or *modules*, for short).

Variables vs. events. A 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

<table>
<tr><td>$privX_P$</td><td>$intfX_P$</td><td>$extlX_P$</td></tr>
<tr><td colspan="2">$ctrX_P$</td><td></td></tr>
<tr><td></td><td colspan="2">$obsX_P$</td></tr>
<tr><td colspan="3">X_P</td></tr>
</table>

Table 1: Module variables

variables. For example, 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 $s[tick] \neq t[tick]$.[1]

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*.

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 visible to the environment, and therefore called *observable*. The set of observable variables of P is denoted $obsX_P$. An *observation* of P is a valuation for the variables in $obsX_P$. The various classes of module variables and their relationships are summarized in Table 1. The distinction between private, interface, and external variables is similar to the distinction between internal, output, and input events in the formalism of I/O automata [21].

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 [20, 10, 17, 22]): Either the system performs an update, or the environment performs an update. Interleaving models usually distinguish only between private variables, which can be updated by the system alone, and shared variables, which can be updated by both the system and the environment. This is a natural style for modeling asynchronous communication via a shared memory.

Observable asynchrony (I/O automata [21]): Either the system updates the controlled variables, or the environment updates the external variables and the system updates the private variables in response. This is a natural style for modeling asynchronous communication via events or messages.

Atomic synchrony (Mealy machines [23, 18]; CSP rendezvous [16, 24]): The system and the environment simultaneously update variables in an interdependent fashion. This is a natural style for modeling synchronous communication via events or messages.

Nonatomic synchrony (synchronous programming languages [8, 7, 9, 14]): 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. This is a natural style for modeling when a computation of arbitrary duration can be synchronized with a single event.

[1] If s is a state and x is a variable in the domain of s, we write $s[x]$ for the value assigned by s to x.

The first two options lead to nonblocking communication, as the system puts no constraints on what the environment can do, nor on the speed at which the environment performs its updates. Nonblocking communication supports compositional reasoning with respect to a trace semantics. However, the inherently asynchronous nature of the communication in these two options render them unsatisfactory for modeling intrinsically synchronous systems such as hardware. The third option leads to the possibility of deadlocks in communication, and the fourth option may lead to the possibility of nonterminating computation within a single round. Both prospects raise difficulties for achieving a compositional trace semantics.

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. Third, the atoms are partially ordered. If atom A precedes atom B in the partial order, 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.

Thus, while synchronous languages, such as ESTEREL [8] and SIGNAL [7], permit the *dynamic* scheduling of subrounds at run time, atoms can be scheduled *statically* at compile time by linearizing the partial order on the atoms. This approach, while more restrictive, has also been taken in recent ESTEREL compilers which perform compile-time safety checks to reject programs that may lead at run-time to nonterminating computations within a round. However, while the emphasis in synchronous languages is on efficient programming of reactive systems, the emphasis in reactive modules is on modeling. This, in particular, has led us to the following two design choices in addition to nonblocking communication. First, reactive modules support nondeterminism and temporal abstraction operators, both of which are convenient for describing high-level or incomplete designs. Specifically, nondeterminism can be used for modeling asynchronous processes that proceed at independent speeds; and temporal abstraction can be used for modeling, at a high level, apparent circular dependencies that can be resolved at a lower level. Second, reactive modules are based on the notion of explicit state, in order to closely resemble state-transition graphs. The envisioned scenario is the one in which a variety of different programming languages and hardware description languages are translated into reactive modules, and verification is then performed on the resulting modules.

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. Given a set X of unprimed symbols, we write X' for the set of corresponding primed symbols.

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. Given two sets X and Y of variables, 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.

Atoms vs. modules. The controlled variables of a module P are partitioned into atoms; that is, every controlled variable of P is controlled by one and only one atom of P. The initialization round and all update rounds consist of several subrounds, one for the environment and one for each atom. For each atom A, in the A-subround of the initialization round, all variables controlled by A are initialized simultaneously, as defined by an initial action. In the A-subround of each update round, all variables controlled by A are updated simultaneously, as defined by an update action.

Definition 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 $ctrX'_A$.*

In the initialization round, the initial action of the atom A assigns initial values to the controlled variables as a nondeterministic function of the initial values of the awaited variables. In each update round, the update action of A assigns updated values to the controlled variables 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 awaited variables have already been updated. The variable y *awaits* the variable x, written $y \succ_A x$, if $y \in ctrX_A$ and $x \in waitX_A$. Now, we can define reactive modules.

Definition 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 order $A_0, \ldots, A_{k-1}$ of the atoms in $\mathcal{A}_P$ is *consistent* if for all $0 \leq i < j < k$, 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 order of the atoms in $\mathcal{A}_P$.

Module execution. When executing the module P, in each round, first the external variables are assigned arbitrary values of the correct types, and then the atoms in $\mathcal{A}_P$ are executed in an arbitrary consistent order $A_0, \ldots, A_{k-1}$. Specifically, in the initialization round, after the external variables have been initialized, the initial action of A_0 is followed by the initial action of A_1 etc.; and in each update round, after the external variables have been updated, the update action of A_0 is followed by the update action of A_1 etc. In this manner, all awaited values are available when they are needed during the execution, and thus every round can be completed in $k+1$ subrounds —one subround for the environment followed by one subround for each atom. Moreover, all nondeterminism in the completion of a round is caused by the nondeterminism of the environment and by the nondeterminism of individual atoms, not by the order of the atoms.

3 Examples of Reactive Modules

The syntax we use for specifying modules will be comprehensible once a few conventions are explained. Variable declarations are indicated by keywords such as **awaits**, for the awaited variables

module *Not*
  external $in : \mathbb{B}$
  interface $out : \mathbb{B}$
  atom *out* **awaits** *in*
    init update
      $[]\ in' = 0 \ \rightarrow\ out' := 1$
      $[]\ in' = 1 \ \rightarrow\ out' := 0$

module *And*
  external $in_1, in_2 : \mathbb{B}$
  interface $out : \mathbb{B}$
  atom *out* **awaits** in_1, in_2
    init update
      $[]\ in'_1 = 0 \ \rightarrow\ out' := 0$
      $[]\ in'_2 = 0 \ \rightarrow\ out' := 0$
      $[]\ in'_1 = 1 \wedge in'_2 = 1 \ \rightarrow\ out' := 1$

module *Latch*
  external $set, reset : \mathbb{B}$
  interface $out : \mathbb{B}$
  private $state : \mathbb{B}$
  atom *out* **reads** *state*
    update
      $[]\ true \ \rightarrow\ out' := state$
  atom *state* **awaits** $set, reset, out$
    init update
      $[]\ set' = 0 \wedge reset = 0 \ \rightarrow\ state' := out'$
      $[]\ set' = 1 \ \rightarrow\ state' := 1$
      $[]\ reset' = 1 \ \rightarrow\ state' := 0$

Figure 1: Synchronous NOT gate, AND gate, and latch

of an atom, or **private**, for the private variables of a module. The initial and update actions of atoms are specified by the keywords **init** and **update**, followed by nondeterministic guarded commands. The combined keyword **init update** indicates that the guarded command that follows specifies both the initial and update actions. If several guards of a guarded command are true, then one of the corresponding assignments is chosen nondeterministically; if none of the guards are true, then all controlled variables obtain their default values. In the initialization round, the default initial value of a variable is chosen nondeterministically (this is legal only if the variable has finite type). In each update round, the default updated value of a variable is equal to the latched value of the variable (i.e., the value of the variable stays unchanged). The default values are also invoked if a controlled variable does not appear on the left-hand-side of an assignment, or if the initial command is omitted altogether.

3.1 Synchronous circuits

Synchronous circuits are built from logic gates and memory cells that are driven by a sequence of clock ticks. Each logic gate computes a boolean value once per clock cycle, and each memory cell stores a boolean value from one clock cycle to the next. We model each logic gate and each memory cell as a reactive module so that every update round represents a clock cycle. All sequential circuits can be constructed from the building blocks shown in Figure 1.

The module *Not* models a synchronous NOT gate, which takes a boolean input and produces a boolean output. The input is modeled as an external variable, *in*, because it is modified by the environment and visible to the gate. The output is modeled as an interface variable, *out*, because it is modified by the gate and visible to the environment. In the initialization round, the NOT gate waits for the input value to be initialized before computing the initial output value, by negating the initial input value. In each update round, the NOT gate waits for the input value to be updated before computing the next output value, by negating the updated input value.

The module *And* 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 initialized or updated after both external variables in_1 and in_2 have been initialized or 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. In the initialization round, the output of the latch is arbitrary, and the state of the latch is initialized after both inputs have been initialized. If both inputs are initially low, then the initial state of the latch is arbitrary, but equal to the initial output.

History-free variables in verification. Given a module P, a variable x of P is *history-free* if x is not read by any atom of P. Then, the update commands of P can refer only to the updated value x' and not to the latched value x. In synchronous circuits, all variables that represent wires are history-free. Specifically, all variables of Figure 1 except for the latch state, *state*, are history-free. In each round, the possible updated values of a history-free variable x depend only on the latched values of variables that are not history-free, and on the updated values of variables other than x. In this way, history-free variables are analogous to the combinational variables of hardware description languages, the selection variables of COSPAN [18], and the pointwise functions of dataflow languages. Hence, during the verification of a module by explicit search through the state space, the values of history-free variables can be omitted from the search stack. Similarly, during the symbolic verification of a module, the history-free variables can be eliminated using existential quantification in each image-computation step.

3.2 Asynchronous shared-memory programs

As an example of a concurrent program consisting of processes that communicate through read-shared variables, we consider a mutual-exclusion protocol, which ensures that no two processes simultaneously access a common resource. 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

```
module P_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
      [] pc_1 = outCS → pc'_1 := reqCS; x'_1 := x_2
      [] pc_1 = reqCS ∧ (pc_2 = outCS ∨ x_1 ≠ x_2) → pc'_1 := inCS
      [] pc_1 = inCS → pc'_1 := outCS
      [] true →

module P_2
  interface pc_2 : {outCS, reqCS, inCS}; x_2 : B
  external pc_1 : {outCS, reqCS, inCS}; x_1 : B
  atom pc_2, x_2 reads pc_1, pc_2, x_1, x_2
    init
      [] true → pc'_2 := outCS
    update
      [] pc_2 = outCS → pc'_2 := reqCS; x'_2 := ¬x_1
      [] pc_2 = reqCS ∧ (pc_1 = outCS ∨ x_1 = x_2) → pc'_2 := inCS
      [] pc_2 = inCS → pc'_2 := outCS
      [] true →
```

Figure 2: Asynchronous mutual-exclusion protocol

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. Note that each process may sleep for arbitrarily many rounds: nondeterminism is used to ensure that there is no relationship between the execution speeds of the two processes.

Interleaving. Unlike in interleaving models, both processes may modify their variables in the same round. While Peterson's 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. Alternatively, one could describe the complete protocol as a single module containing a single atom whose update action is the union of the update actions of the atoms of Figure 2. The guarded command that specifies a union of actions consists simply of the union of all guarded assignments of the individual actions. This style of describing asynchronous programs as an unstructured collection of guarded assignments is pursued in formalisms such as UNITY [10] and MURφ [12].

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 awaited external variables such as $P_i\_sets\_x\_to\_0$, which is a boolean interface variable of the i-th process that indicates when the process wants to set x to 0. Since all of these variables with the exception of x are history-free, the modeling does not increase the search space during verification. This style of describing write-shared memory makes explicit what happens when several processes write simultaneously to the same location.

3.3 Synchronous message-passing protocols

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 receiver. If the receiver is ready to receive a message, it issues the interface event *ready* to signal its readiness to the sender. The

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

module $Receiver$
  external $transmit : \mathbb{E};\ msg_S : \mathbb{M}$
  interface $ready : \mathbb{E};\ msg_C : \mathbb{M}$
  private $pc : \{receive, consume\};\ done_C : \mathbb{E};\ msg_R : \mathbb{M}$
  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** $pc, ready$
    update
      $[\!]\ pc = receive \ \rightarrow\ ready!$
      $[\!]\ true \ \rightarrow$
  $A\mathrm{Cons}$: **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

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 *A*Cons, 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.

Event variables in verification. Like history-free variables, event variables are also used only for interaction within a round, and their actual values at the beginning of a round are immaterial. In a sense, the values of event variables behave like labels on the transitions of a state-transition graph, unlike the values of other variables, which behave like labels on the states. Consequently, during explicit verification, the values of event variables can be omitted from the search stack, and during symbolic verification, event variables can be eliminated using existential quantification.

4 Semantics of Reactive Modules

The execution of a module results in a trace of observations. Reactive modules are related via a trace semantics: roughly speaking, one module *implements* (or *refines*) another module if all possible traces of the former, more detailed module are also possible traces of the latter, more abstract module.

4.1 The trace language of a module

Let P be a reactive module. As indicated earlier, a state of P is a valuation for the set X_P of module variables. We write Σ_P for the set of states of P.

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 $A \in \mathcal{A}_P$, $(s'[waitX'_A], s'[ctrX'_A]) \in Init_A$.[2] We write $Init_P$ for the set of initial states of the module P. The set $Init_P$ is nonempty. In fact, for every valuation s^e for the external variables of P, there is a nonzero but finite number of initial states s with $s[extlX_P] = s^e$. This is because all initial actions are executable.

For two states s and t of P, the state t is a *successor* of s, written $s \rightarrow_P t$, if t can be obtained from s by executing all update actions of P in a consistent order: for each atom $A \in \mathcal{A}_P$, $(s[readX_A] \cup t'[waitX'_A], t'[ctrX'_A]) \in Update_A$. The binary relation $\rightarrow_P$ over the state space Σ_P is called the *transition relation* of the module P. The transition relation $\rightarrow_P$ is serial (i.e., every state has at least one successor). In fact, for every state s of P, and for every valuation t^e for the external variables of P, there is a nonzero and finite number of states t with $s \rightarrow_P t$ and $t[extlX_P] = t^e$. This is because all update actions are executable. In other words, a module does not constrain the behavior of the external variables and interacts with its environment in a nonblocking way.

In this way, the module P defines a state-transition graph with the state space Σ_P, the initial states $Init_P$, and the transition relation $\rightarrow_P$. The initialized paths of this graph are called the trajectories

[2]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 $X \cup Y$.

of the module: a *trajectory* of P is a finite sequence $s_0 \ldots s_n$ of states of P 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. A state s of P is *reachable* if there is a trajectory of P whose last state is s. If $\overline{s} = s_0 \ldots s_n$ is a trajectory of P, then the corresponding sequence $\overline{s}[obsX_P] = s_0[obsX_P] \ldots s_n[obsX_P]$ of observations is called a *trace* of P. Thus, a trace records the sequence of observations that may result from executing the module for finitely many steps. The *trace language* of the module P, denoted L_P, is the set of traces of P. By definition, every prefix of a trajectory is also a trajectory, and hence, every prefix of a trace is also a trace. Since the set of initial states is nonempty, and the transition relation is serial, every trajectory of a module, and hence also every trace, can be extended.

Proposition 1 *For every module P, the trace language L_P is prefix-closed and contains traces of arbitrary length.*

It follows that a module cannot deadlock. In modeling, therefore, a deadlock situation must be represented by a special state with a single outgoing transition back to itself.

4.2 The implementation preorder between modules

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 (there cannot be any await dependencies between external variables and other variables).

Definition 3 [**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 observable variables x of Q and all interface variables y of Q, if $y \succ_Q x$, then $y \succ_P x$; and (4) if $\overline{s}$ is a trace of P, then the projection $\overline{s}[obsX_Q]$ is a trace of Q.*

The first three conditions ensure that the compatibility constraints imposed by P on its environment are at least as strong as those imposed by Q. The fourth condition is conventional trace containment. Intuitively, if $P \preceq Q$, then the module P is *as detailed as* the module Q: the implementation P has possibly more interface and external variables than the specification Q; some external variables of Q may be interface variables of P, and thus are more constrained in P; the implementation P has possibly more await dependencies among its observable variables than the specification Q; and P has possibly fewer traces than Q, and thus more constraints on its execution. It is easy to check that every module P implements itself, and that if a module P implements another module Q, which, in turn, implements a third module R, then P also implements R.

Proposition 2 *The implementation relation $\preceq$ is a preorder on modules (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 on modules. The *meaning* of a module P is the $\cong$-equivalence class of P.

4.3 Special classes of atoms and modules

Combinational vs. sequential atoms. An atom A is *combinational* if it has (1) no read variables and (2) identical initial and update actions: $readX_A = \emptyset$, and $Init_A = Update_A$. In each update round, the updated values of the controlled variables of a combinational atom depend only on the updated values of other variables, and not on any latched values. Furthermore, a combinational atom cannot distinguish between the initialization round and later rounds. If an atom is not combinational, then it is called *sequential*. For example, in Figure 1, the atoms of the modules *Not* and *And* and the atom that controls the variable *state* of the module *Latch* are combinational; the atom that controls the variable *out* of *Latch* is sequential. Note that a combinational atom may control some variables that are not history-free, and an atom may be sequential despite controlling only history-free variables. The two cases apply to the two atoms of the module *Latch*.

For further illustration of the difference between combinational and sequential atoms, consider the following example. Given two variables x and y of the same type, we want x to duplicate the behavior of y. The combinational atom

ACombCopy: **atom** x **awaits** y
  init update
    $[\!]$ $true \rightarrow x' := y'$

copies y into x without delay, and ensures that both x and y have the same value at the end of each round. The sequential atom

ASeqCopy: **atom** x **reads** y
  update
    $[\!]$ $true \rightarrow x' := y$

copies y into x with a delay of one round. In each update round, x is assigned the value of y at the beginning of the round (the initial command is irrelevant for the purposes of this example).

Lazy vs. eager atoms. An atom *sleeps* in an update round if the values of all controlled 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, $(s[readX_A] \cup t'[waitX'_A], s'[ctrX'_A]) \in Update_A$. A sufficient syntactic condition for laziness is the presence of the guarded assignment "$true \rightarrow$" in the update command, which leaves the values of all controlled variables unchanged. For example, the atoms of the modules P_1 and P_2 from Figure 2 are lazy. Typically, lazy atoms are nondeterministic and sequential, with all controlled variables being read in order to keep their values unchanged.

If an atom is not lazy, then it is called *eager*. Both atoms ACombCopy and ASeqCopy are eager. In the first case, all updates of x follow immediately, within the same round, the corresponding updates of y; in the second case, the updates of x are delayed by exactly one round. By contrast, the lazy atom

ALazyCopy: **atom** x **reads** x **awaits** y
  init update
    $[\!]$ $true \rightarrow x' := y'$
    $[\!]$ $true \rightarrow$

copies y into x at arbitrary times. In each update round, either the value of x stays unchanged, or it is set to the updated value of y. Consequently, some values of y may not be copied into x.

Event-driven vs. round-driven atoms. In each update round, an atom can notice changes in the values of awaited variables. If an awaited variable is also read, then the atom can directly compare the latched value with the updated value. If an awaited variable is not read, then the atom can remember the latched value from the previous round, by storing it in a controlled variable, and still compare the latched value with the updated value. Therefore, each change in the value of an awaited variable is an observable event. An X-atom A is *event-driven* if it may sleep in every update round in which no observable event occurs; that is, the atom may sleep whenever the values of all awaited variables stay unchanged: for all valuations s for X, $(s[readX_A] \cup s'[waitX'_A], s'[ctrX'_A]) \in Update_A$. While the progress of a lazy atom cannot be enforced at all, the progress of an event-driven atom A can be enforced by other atoms that modify awaited variables. However, the progress of an event-driven atom cannot be enforced solely by the expiration of rounds. Hence, if an atom is not event-driven, then it is called *round-driven*.

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. Second, every lazy atom is event-driven. For example, all atoms of the modules *Sender* and *Receiver* from Figure 3 are event-driven. Third, every combinational atom is event-driven. This is because if the awaited variables do not change in an update round, then the combinational atom may compute the same values for the controlled variables as in the previous round. For example, while the sequential atom ASeqCopy is round-driven, the behavior of the combinational atom ACombCopy can be alternatively defined by the atom

AEventCopy: **atom** x **reads** x, y **awaits** y
  init
    $[\!]$ $true \rightarrow x' := y'$
  update
    $[\!]$ $y' \neq y \rightarrow x' := y'$

because the value of x needs to be modified only when the value of y changes. This explicitly event-driven specification of immediate copying, however, reads both x and y, and is no longer combinational (y is read to check if the value of y changes, and x is read to keep the value of x unchanged).

Asynchronous vs. synchronous modules. A module *stutters* in an update round if the values of all interface variables stay unchanged. Asynchronous modules are defined so that they may stutter in every update round.

Definition 4 [**Asynchrony**] *A module P is* asynchronous *if all interface variables of P are controlled by lazy atoms. Otherwise, P is a* synchronous *module.*

It follows that the environment cannot enforce the observable progress of an asynchronous module. While an asynchronous module can privately record all changes in the values of external variables, all updates of interface variables proceed at a speed that is independent of the environment speed. For example, the modules P_1 and P_2 from Figure 2 are asynchronous.

Round-insensitive vs. round-sensitive modules. 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. Round-insensitive modules are defined so that they may sleep in every update round in which the environment stutters.

Definition 5 [**Round-insensitivity**] *A module P is* round-insensitive *if all atoms of P are event-driven. Otherwise, P is a* round-sensitive *module.*

module *RoundCount*
  interface *count*: $\mathbb{N}$
  atom *count* **reads** *count*
    init
      $[\!]\ true \ \rightarrow\ count' := 0$
    update
      $[\!]\ true \ \rightarrow\ count' := count + 1$

module *EventCount*
  external *tick*: $\mathbb{E}$
  interface *count*: $\mathbb{N}$
  atom *count* **reads** *count*, *tick* **awaits** *tick*
    init
      $[\!]\ true \ \rightarrow\ count' := 0$
    update
      $[\!]\ tick? \ \rightarrow\ count' := count + 1$

module *AsyncCount*
  interface *count*: $\mathbb{N}$
  atom *count* **reads** *count*
    init
      $[\!]\ true \ \rightarrow\ count' := 0$
    update
      $[\!]\ true \ \rightarrow\ count' := count + 1$
      $[\!]\ true \ \rightarrow$

Figure 4: Three counters

It follows that the trace language of a round-insensitive module P is closed under the insertion of stutter steps: if $a_0 \ldots a_n$ is a trace of P, then so are the observation sequences $a_0 \ldots a_i a_i \ldots a_n$ for all $0 \leq i \leq n$. For example, the modules *Sender* and *Receiver* from Figure 3 are round-insensitive.

Asynchrony and round-insensitivity are independent: a module may be synchronous and round-sensitive, asynchronous and round-sensitive, synchronous and round-insensitive, or asynchronous and round-insensitive. 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 *EventCount* is incremented with every occurrence of the external event *tick*, and the round-insensitive asynchronous counter *AsyncCount* is incremented nondeterministically.

5 Spatial Operations on Reactive Modules

We create complex modules from simple modules using the three spatial operations of variable renaming, parallel composition, and variable hiding, and the two temporal operations of round abstraction and triggering. Spatial operations manipulate the variables of a module, leaving the underlying notion of round fixed; temporal operations manipulate what happens during a round, leaving the variables fixed. We discuss the three spatial operations in this section, and the two temporal operations in the next.

All five operations f on modules are compositional in the sense that the equivalence relation $\cong$ is a congruence with respect to f: for all modules P and Q, if $P \preceq Q$, then $f(P) \preceq f(Q)$. Thus, if we prove that module P is $\cong$-equivalent to module Q, then P can be substituted for Q in every context without affecting the meaning of the complex module. Furthermore, in order to prove that $f(P)$ implements $f(Q)$, it suffices to prove that P implements Q. In this way, reasoning about complex modules can be reduced to reasoning about simpler submodules.

5.1 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. Henceforth, whenever we write $P[x := y]$, we assume that x and y have the same type, and that y is not a module variable of P. We make liberal use of notation such as $P[x, y := y, x]$ for $P[x := z][y := x][z := y]$. We furthermore assume that for any two modules, a private variable of one module is not a module variable of the other module. This can always be achieved by renaming private variables, which does not change the meaning of a module. It is obvious that the renaming operation is compositional: for all modules P and Q, if $P \preceq Q$, then $P[x := y] \preceq Q[x := y]$.

5.2 Parallel composition

The composition operation combines two modules into a single module whose behavior captures the interaction between the two component modules. 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 the 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 6 [**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 $extlX_{P\|Q} = (extlX_P \cup extlX_Q) \setminus intfX_{P\|Q}$ 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 the modules P and Q are compatible. The composition operation on modules is commutative and associative. We therefore omit parentheses when writing $P\|Q\|R$.

Parallel composition behaves like language intersection. This is captured by the following proposition, which asserts that the traces of a compound module are completely determined by the traces of the component modules. In particular, if P and Q have identical observations, then $L_{P\|Q} = L_P \cap L_Q$.

Proposition 3 *Let P and Q be two compatible modules, and let $\overline{a}$ be a finite sequence of observations of the compound module $P\|Q$. Then, $\overline{a}$ belongs to the language $L_{P\|Q}$ iff the projection $\overline{a}[obsX_P]$ belongs to L_P and the projection $\overline{a}[obsX_Q]$ belongs to L_Q.*

It follows that, up to projection, the trace language of a compound module is a subset of the trace language of each component. Hence, the composition of two modules creates a module that is equally or more detailed than its components. This is captured by the first part of the following proposition. The second part asserts that the composition operation is compositional.

Proposition 4 *Let P, Q, and R be three modules such that P and R are compatible. Then (1) $P\|R \preceq P$, and (2) $P \preceq Q$ implies $P\|R \preceq Q\|R$.*

Proof. Part (1) follows from Proposition 3. For part (2), consider two modules P and Q such that $P \preceq Q$, and a module R that is compatible with P. Then R is also compatible with Q. The definition of implementation has four conditions. The first three conditions are immediate. The fourth condition is trace containment. Let $\overline{a}$ be a trace of $P\|R$. By Proposition 3, the projection $\overline{a}[obsX_P]$ is a trace of P, and the projection $\overline{a}[obsX_R]$ is a trace of R. Since $P \preceq Q$, the projection $\overline{a}[obsX_Q]$ is a trace of Q. Again by Proposition 3, the projection $\overline{a}[obsX_{Q\|R}]$ is a trace of $Q\|R$. ■

It follows that, in order to prove that a complex compound module $P_1\|P_2$ (with a large state space) implements a simpler compound module $Q_1\|Q_2$ (with a small state space), it suffices to prove (1) P_1 implements Q_1 and (2) P_2 implements Q_2. We call this the *compositional proof rule* for reactive modules. It is valid, because parallel composition and implementation behave like language intersection and language containment, respectively.

Assume-guarantee reasoning. While the compositional proof rule decomposes the verification task of proving implementation between compound modules into subtasks, it may not always be applicable. In particular, P_1 may not implement Q_1 for all environments, but only if the environment behaves like P_2, and vice versa. For such cases, an assume-guarantee proof rule is needed [25, 13, 2, 4]. The *assume-guarantee proof rule* for reactive modules asserts that in order to prove that $P_1\|P_2$ implements $Q_1\|Q_2$, it suffices to prove (1) $P_1\|Q_2$ implements Q_1, and (2) $Q_1\|P_2$ implements Q_2. Both proof obligations (1) and (2) typically involve smaller state spaces than the original proof obligation, because the complex compound module $P_1\|P_2$ usually has the largest state space involved. The assume-guarantee proof rule is circular; unlike the compositional proof rule, it does not simply follow from the fact that parallel composition and implementation behave like language intersection and language containment. Rather the proof of the validity of the assume-guarantee proof rule proceeds by induction on the length of traces. For this, it is crucial that every trace of a module can be extended.

Proposition 5 *Let P_1 and P_2 be two compatible modules, and let Q_1 and Q_2 be two compatible modules such that every external variable of $Q_1\|Q_2$ is an observable variable of $P_1\|P_2$. If $P_1\|Q_2 \preceq Q_1$ and $Q_1\|P_2 \preceq Q_2$, then $P_1\|P_2 \preceq Q_1\|Q_2$.*

Proof. Consider four modules P_1, P_2, Q_1, and Q_2 such that (1) P_1 and P_2 are compatible, (2) Q_1 and Q_2 are compatible, (3) every external variable of $Q_1\|Q_2$ is an observable variable of $P_1\|P_2$,

(4) $P_1\|Q_2 \preceq Q_1$, and (5) $Q_1\|P_2 \preceq Q_2$. We wish to establish that $P_1\|P_2 \preceq Q_1\|Q_2$. The definition of implementation has four conditions. Let us consider these four proof obligations one by one.

Condition 1: every interface variable of $Q_1\|Q_2$ is an interface variable of $P_1\|P_2$. Let x be an interface variable of $Q_1\|Q_2$. Without loss of generality, assume that x is an interface variable of Q_1. Assumption (4) implies that x is an interface variable of $P_1\|Q_2$. Assumption (2) implies that x is not an interface variable of Q_2. It follows, from the definition of parallel composition, that x is an interface variable of P_1, and hence, of $P_1\|P_2$.

Condition 2: every external variable of $Q_1\|Q_2$ is an observable variable of $P_1\|P_2$. This is assumption (3).

Condition 3: for all observable variables x of $Q_1\|Q_2$ and all interface variables y of $Q_1\|Q_2$, if $y \succ_{Q_1\|Q_2} x$, then $y \succ_{P_1\|P_2} x$. We show the stronger claim that for interface variables $z_1, \ldots, z_i$ of $P_1\|P_2$, interface variables $y_1, \ldots, y_j$ of $P_1\|P_2$ or $Q_1\|Q_2$, and observable variables x of $Q_1\|Q_2$, if

$$y = z_1 \succ_{P_1\|P_2} \cdots\ z_i \succ_{P_1\|P_2} y_1 \succ_{R_1} \cdots\ y_j \succ_{R_j} x,$$

where $R_l \in \{P_1, P_2, Q_1, Q_2\}$ for all $1 \leq l \leq j$, then $y \succ_{P_1\|P_2} x$. Condition (3) then follows from the special case that $i = 0$, and hence, $y = y_1$.

In the claim, since the relation $\succ_{P_1\|P_2}$ is acyclic by assumption (1), the variables $z_1, \ldots, z_i$ are all pairwise distinct. Therefore $0 \leq i \leq n$, where n is the number of interface variables of $P_1\|P_2$. We prove the claim by decreasing induction on i: consider $i \in \{0, \ldots, n\}$, assume as induction hypothesis that $i < n$ implies the claim holds for $i + 1$, and show the claim for i. If $j = 0$, then $y \succ_{P_1\|P_2} x$ by the transitivity of $\succ_{P_1\|P_2}$. If $j \geq 1$, then there are four possibilities for R_1. If $R_1 \in \{P_1, P_2\}$, then y_1 is an interface variable of $P_1\|P_2$. The acyclicity of $\succ_{P_1\|P_2}$ implies that $i < n$, and the claim follows by induction hypothesis (choose $z_{i+1} = y_1$). If $R_1 = Q_1$, then y_1 is an interface variable of Q_1, and therefore by assumption (2), it is not an interface variable of Q_2. Since $y_1 \succ_{Q_1} y_2$, from assumption (4) it follows that $y_1 \succ_{P_1\|Q_2} y_2$. Since y_1 is not an interface variable of Q_2, it is an interface variable of P_1, and therefore $y_1 \succ_{P_1} u_1 \succ_{R'_1} \cdots\ u_k \succ_{R'_k} y_2$, for interface variables $u_1, \ldots, u_k$ of $P_1\|Q_2$, and $R'_l \in \{P_1, Q_2\}$ for all $1 \leq l \leq k$. Again, the acyclicity of $\succ_{P_1\|P_2}$ implies that $i < n$, and the claim follows by induction hypothesis (choose $z_{i+1} = y_1$). The final case, $R_1 = Q_2$, is symmetric to the previous case.

Condition 4: if $\overline{a}$ is a trace of $P_1\|P_2$, then the projection of $\overline{a}$ to the observable variables of $Q_1\|Q_2$ is a trace of $Q_1\|Q_2$. In the following, for simplicity we omit the explicit use of projections. For example, if X is a superset of $obsX_P$, if $\overline{a}$ is a sequence of valuations for X, when we refer to $\overline{a}$ as a trace of P, what we mean is that the projection $a[obsX_P]$ is a trace of P.

We need to define some additional concepts. Given a module P, a set X of variables is *await-closed* for P if for all observable variables x and y of P, if $y \succ_P x$ and $y \in X$, then $x \in X$. For an await-closed set X, the pair $(\overline{a}, b)$ consisting of a trace $\overline{a}$ of P and a valuation b for X is an *X-partial trace* of P if there exists an observation c of P such that (1) $c[X \cap obsX_P] = b[obsX_P]$, and (2) $\overline{a}c$ is a trace of P. Thus, partial traces are obtained by executing several complete rounds followed by a partial round, in which only some of the atoms are executed. The following facts about partial traces follow from the definitions.

(A) If $P \preceq Q$ and X is an await-closed set of variables for P, then X is an await-closed set of variables for Q. If $P \preceq Q$ and $(\overline{a}, b)$ is an X-partial trace of P, then $(\overline{a}, b)$ is an X-partial trace of Q. Thus, trace containment is equivalent to containment of partial traces.

(B) The partial traces of a compound module are determined by the partial traces of the component modules: for every await-closed set X for $P\|Q$, every sequence $\overline{a}$ of observations of $P\|Q$, and every valuation b for X, the pair $(\overline{a}, b)$ is an X-partial trace of $P\|Q$ iff it is an X-partial trace of both P and Q.

(C) If $(\overline{a}, b)$ is an X-partial trace of P, and c is a valuation for a set Y of variables of P, which is disjoint from both X and $intfX_P$, then $(\overline{a}, b \cup c)$ is an $(X \cup Y)$-partial trace of P. This property is due the nonblocking nature of modules.

Let $X_1, \ldots, X_m$ be a partition of $obsX_{P_1\|P_2}$ into disjoint subsets such that (1) each X_i either contains only external variables of $P_1\|P_2$, or contains only interface variables of P_1, or contains only interface variables of P_2, and (2) if $y \succ_{P_1\|P_2} x$ and $y \in X_i$, then $x \in X_j$ for some $j < i$. Define $Y_0 = \emptyset$, and for all $0 \leq i < m$, define $Y_{i+1} = Y_i \cup X_i$. Each set Y_i is await-closed for $P_1\|P_2$. For all $0 \leq i \leq m$, let L_i be the set of Y_i-partial traces of $P_1\|P_2$, and let $L = \cup_{0 \leq i \leq m} L_i$. We define the following order $<$ on the partial traces in L: for $i < m$, if $(\overline{a}, b) \in L_i$ and $(\overline{a}, c) \in L_{i+1}$ and $c[Y_i] = b$, then $(\overline{a}, b) < (\overline{a}, c)$; for $i = m$, if $(\overline{a}, b) \in L_i$, then $(\overline{a}, b) < (\overline{a}b, \emptyset)$. Clearly, the order $<$ is well-founded. We prove by well-founded induction with respect to $<$ that for all $0 \leq i \leq m$, every partial trace in L_i is a Y_i-partial trace of $Q_1\|Q_2$. Then, the case $i = 0$ implies that every trace of $P_1\|P_2$ is also a trace of $Q_1\|Q_2$.

Consider $(\overline{a}, \emptyset)$ in L_0. If $\overline{a}$ is the empty trace, then $(\overline{a}, \emptyset)$ is a trace of all modules. Otherwise, $\overline{a} = \overline{b}c$ for some observation sequence $\overline{b}$ and observation c of $P_1\|P_2$. Then $(\overline{b}, c)$ is a Y_m-partial trace of $P_1\|P_2$, and $(\overline{b}, c) < (\overline{a}, \emptyset)$. By induction hypothesis, $(\overline{b}, c)$ is a Y_m-partial trace of $Q_1\|Q_2$, and hence, $(\overline{a}, \emptyset)$ is a Y_0-partial trace of $Q_1\|Q_2$.

Consider $(\overline{a}, b)$ in L_{i+1} for some $0 \leq i < m$. Let $c = b[Y_i]$. Then $(\overline{a}, c)$ is a Y_i-partial trace of $P_1\|P_2$, and $(\overline{a}, c) < (\overline{a}, b)$. By induction hypothesis, $(\overline{a}, c)$ is a Y_i-partial trace of $Q_1\|Q_2$. By fact (B) about partial traces, $(\overline{a}, c)$ is a Y_i-partial trace of both Q_1 and Q_2. Consider $Y_{i+1} = Y_i \cup X_i$. Without loss of generality, assume that X_i contains no interface variables of P_2, and hence, by assumptions (2) and (5), no interface variables of Q_2. By fact (C) about partial traces, the Y_i-partial trace $(\overline{a}, c)$ of Q_2 can be extended with any valuation for X_i. In particular, $(\overline{a}, b)$ is a Y_{i+1}-partial trace of Q_2. Since $(\overline{a}, b) \in L_{i+1}$, by fact (B) about partial traces, $(\overline{a}, b)$ is a Y_{i+1}-partial trace of P_1, and therefore also of $P_1\|Q_2$. From assumption (4) and fact (A) about partial traces, it follows that $(\overline{a}, b)$ is a Y_{i+1}-partial trace of Q_1. Hence by fact (B) about partial traces, $(\overline{a}, b)$ is a Y_{i+1}-partial trace of $Q_1\|Q_2$. ∎

5.3 Variable hiding

The hiding of interface variables allows us to construct module abstractions of varying degrees of detail. For instance, 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 compound module.

Definition 7 [**Hiding**] *Given a module P and a variable x, by* hiding x *in* P *we obtain the module* **hide** x **in** P. *If x is an interface variable of P, then* **hide** x **in** P *has the set $privX_P \cup \{x\}$ of private variables, the set $intfX_P \backslash \{x\}$ of interface variables, the set $extlX_P$ of external variables, and the set $\mathcal{A}_P$ of atoms. If x is not an interface variable of P, then* **hide** x **in** P *is identical to P.*

We write **hide** x_1, x_2 **in** P short for the module **hide** x_1 **in** (**hide** x_2 **in** P), which is identical to the module **hide** x_2 **in** (**hide** x_1 **in** P). The hiding of a variable creates in a module that is equally

module *SendRecSpec*
  interface $msg_P, msg_C : \mathbb{M}$
  private $pc\colon \{in\_sync, cons\_ahead, prod\_ahead\};\ msg_W\colon \mathbb{M}$
  atom pc, msg_P, msg_W, msg_C **reads** pc, msg_P, msg_W
    init
      $[]\ true \ \rightarrow\ pc' := cons\_ahead$
    update
      $[]\ pc = cons\_ahead \ \rightarrow\ msg'_P := \mathbb{M};\ pc' := in\_sync$
      $[]\ pc = in\_sync \ \rightarrow\ msg'_C := msg_P;\ msg'_P := \mathbb{M};\ pc' := in\_sync$
      $[]\ pc = in\_sync \ \rightarrow\ msg'_W := msg_P;\ msg'_P := \mathbb{M};\ pc' := prod\_ahead$
      $[]\ pc = in\_sync \ \rightarrow\ msg'_C := msg_P;\ pc' := cons\_ahead$
      $[]\ pc = prod\_ahead \ \rightarrow\ msg'_C := msg_W;\ pc' := in\_sync$
      $[]\ true \ \rightarrow$

Figure 5: Asynchronous message-passing specification

or less detailed, and the hiding operation is compositional. Both facts are stated in the following proposition.

Proposition 6 *For all modules P and Q, and every variable x, (1) $P \preceq$ (**hide** x **in** P), and (2) if $P \preceq Q$, then* (**hide** x **in** P) $\preceq$ (**hide** x **in** Q).

From synchrony to asynchrony. Hiding preserves asynchrony, round-sensitivity, 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 $tick\colon \mathbb{E}$
  atom *tick* **reads** *tick*
    update
      $[]\ true \ \rightarrow\ tick!$
      $[]\ true \ \rightarrow$

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

AsyncCount $\cong$ **hide** *tick* **in** (*EventCount* $\|$ *Clock*)

For a more elaborate example, recall the synchronous message-passing protocol from Figure 3:

module *SendRec* $=$ *Sender* $\|$ *Receiver*

After hiding the communication events, so that only the streams of produced messages (msg_P) and consumed messages (msg_C) remain visible, we obtain an asynchronous module:

module *SendRecImpl* $=$ **hide** $ready, transmit, msg_S$ **in** *SendRec*

module *BehavOr*
  external $in_1, in_2 : \mathbb{B}$
  interface $out : \mathbb{B}$
  atom out **awaits** in_1, in_2
    init update
      $[\!]\ in_1' = 0 \ \rightarrow\ out' := 0$
      $[\!]\ in_2' = 0 \ \rightarrow\ out' := 0$
      $[\!]\ in_1' = 1 \vee in_2' = 1 \ \rightarrow\ out' := 1$

module *StructOr* =
  hide z_1, z_2, z_3 **in**
    $\|\ And[in_1, in_2, out := z_1, z_2, z_3]$
    $\|\ Not[in, out := in_1, z_1]$
    $\|\ Not[in, out := in_2, z_2]$
    $\|\ Not[in, out := z_3, out]$

Figure 6: Two definitions of a synchronous OR gate

The module *SendRecImpl* implements the asynchronous module *SendRecSpec* of Figure 5, which contains a single lazy atom:

$$SendRecImpl \ \preceq\ SendRecSpec$$

The module *SendRecSpec* specifies the class of sliding-window protocols with window size 2: the stream of consumed messages results from delaying the stream of produced messages such that at any point, at most two produced messages have not yet been consumed. If, as initially, $pc = cons\_ahead$, then the latest produced message has already been consumed, so the consumer waits and the producer works on producing the next message; if $pc = in\_sync$, then the consumer works on consuming the latest produced message, and the producer works on producing the next message; if $pc = prod\_ahead$, then the previously produced message, which is stored in msg_W, has not yet been consumed, so the consumer works on that message and the producer, constrained by the window size 2, waits. While *SendRecImpl* implements the sliding-window specification using synchronous handshaking, alternatively, *SendRecSpec* could be implemented by sender and receiver processes that communicate via asynchronous handshaking.

5.4 Spatial scaling

The three operations of renaming, composition, and hiding allow us to construct a space hierarchy of modules. We illustrate this on the example of synchronous circuits.

Figure 6 shows two module definitions for a synchronous OR gate. The module *BehavOr* specifies the input-output behavior of an OR gate similar to the definition of the synchronous AND gate from Figure 1. The module *StructOr* builds an OR gate from an AND gate and three inverters. First, we rename the variables of the modules from Figure 1 that define synchronous AND and NOT gates in order to create three instances of a NOT gate and connect, for example, the output of the AND gate with the input of the third NOT gate. Second, we compose the four modules representing the AND gate and the three NOT gates. Third, we hide the variables that represent internal wires, for example, the wire z_3 that connects the output of the AND gate with the input of the third NOT gate. It is easy to check that the two modules *BehavOr* and *StructOr* are $\cong$-equivalent; in particular, they have the same traces.

Using the definitions of synchronous gates and latches from Figure 1, and the three operations of renaming, composition, and hiding, we can build sequential circuits whose clock cycles correspond to rounds. As an example, we design a three-bit binary counter. The counter takes two boolean inputs, represented by the external variables *start* and *inc*, for starting and incrementing the counter. The counter value ranges from 0 to 7, and is represented by three bits. We do not make any assumption about the initial counter value. A start command resets the counter value to 0 and overrides any increment command that is issued in the same round. An increment command increases the counter value by 1. If the counter value is 7, the increment command changes the counter value to 0. In each round, the counter issues its value as output —the low bit on the interface variable out_0, the middle bit on the interface variable out_1, and the high bit on the interface variable out_2.

Figure 7 shows a possible design of the three-bit counter from three one-bit counters. This design is defined by the module *Sync3BitCounter* of Figure 8 (for clarity, we sometimes annotate both component and compound modules with the names of the observable variables). Note that $carry_0$ waits for both *start* and *inc*, that $carry_1$ waits for $carry_0$, and that $carry_2$ waits for $carry_1$. It follows that all three bits of the counter are updated within a single round.

By identifying each clock cycle with a round, we cannot model combinational loops, which would result in cyclic await dependencies. Consider the module *UselessTransLatch* from Figure 9, which models a transparent latch: in each round in which the control input *clk* is true, the data input *in* is without delay propagated to the output *out*; and in each round in which the input *clk* is false, the output *out* stays unchanged. The module *IllegalLoop* composes two transparent latches. Since in each round, the control inputs of the two latches are complementary, all data dependencies can be resolved dynamically, during execution. Our definition of await dependencies, however, is static, and therefore value-independent. Hence, *IllegalLoop* is not a legal module: $latch_1 \succ latch_2$ and $latch_2 \succ latch_1$. In the next section, we will present a legal model for this circuit. We will use several rounds to model a single clock cycle, and then collapse these rounds into a single round.

6 Temporal Operations on Reactive Modules

Each module defines what happens during a round. The notion of round is global: when two modules are composed, the rounds of both component modules are taken to overlap perfectly in time, none being shorter or longer than the other. Throughout the operations of renaming, composition, and hiding, the notion of round stays unchanged: a complex module has the same round as each of its submodules. Sometimes it is convenient, however, to change the notion of what constitutes a round. For example, what happens during a round of a complex module may best be defined by what happens during several consecutive rounds of a submodule. For this purpose, we introduce the operations of round abstraction and triggering.

6.1 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. This can be done by applying the abstraction operator **next**. Intuitively, given a subset Y of the interface variables of a module P, the module **next** Y **for** P

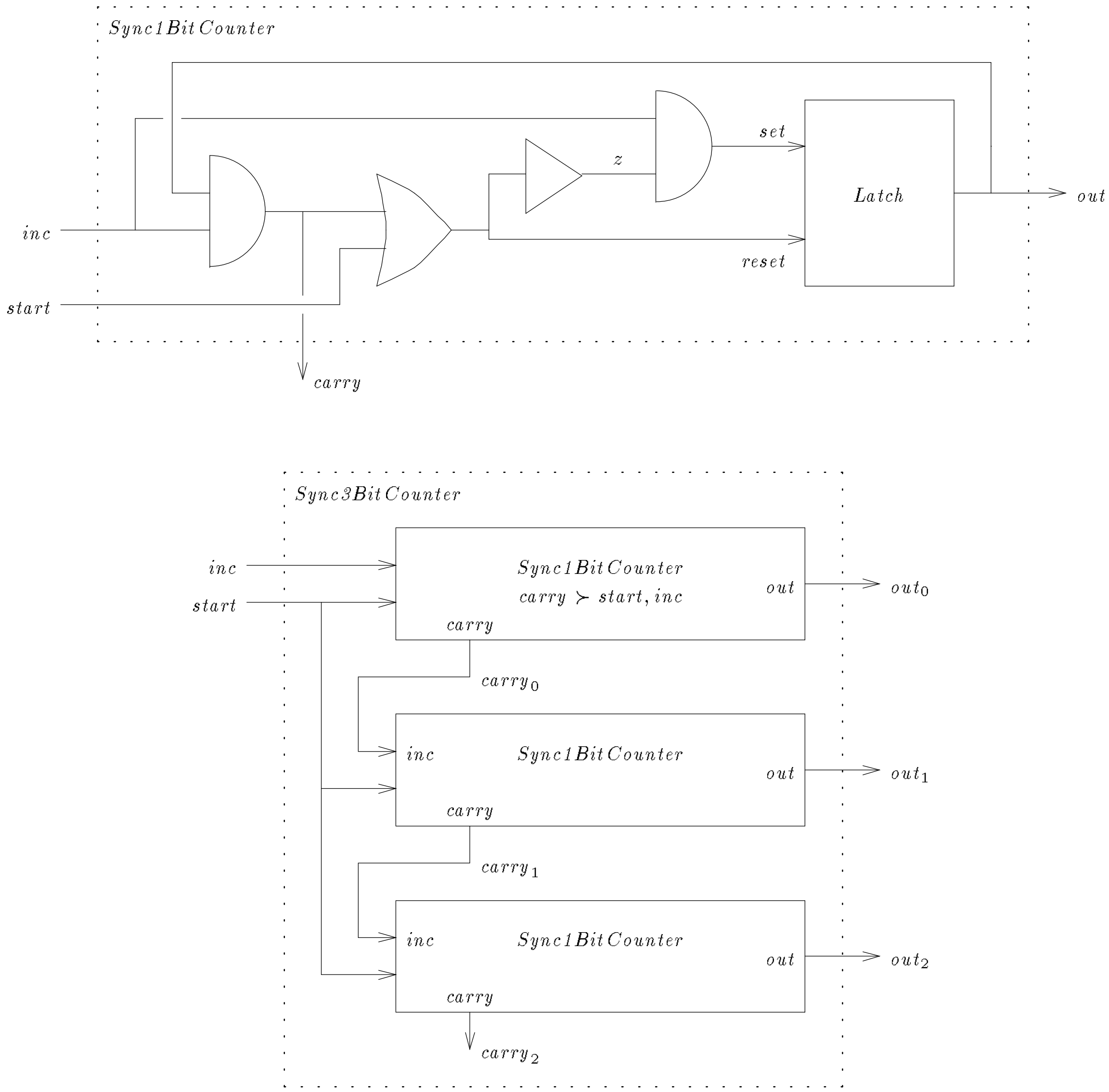

Figure 7: Block diagram for a three-bit binary counter

module *Sync1BitCounter* =
  —**external** $start, inc$
  —**interface** $out, carry$
  hide $set, reset, z$ **in**
    $\|\ Latch[set, reset, out]$
    $\|\ And[in_1, in_2, out := out, inc, carry]$
    $\|\ Or[in_1, in_2, out := carry, start, reset]$
    $\|\ Not[in, out := reset, z]$
    $\|\ And[in_1, in_2, out := inc, z, set]$

module *Sync3BitCounter* =
  —**external** $start, inc$
  —**interface** out_0, out_1, out_2
  hide $carry_0, carry_1, carry_2$ **in**
    $\|\ Sync1BitCounter[start, inc, out, carry := start, inc, out_0, carry_0]$
    $\|\ Sync1BitCounter[start, inc, out, carry := start, carry_0, out_1, carry_1]$
    $\|\ Sync1BitCounter[start, inc, out, carry := start, carry_1, out_2, carry_2]$

Figure 8: Three-bit binary counter

module *UselessTransLatch*
  external $in, clk : \mathbb{B}$
  interface $out : \mathbb{B}$
  atom out **reads** out **awaits** in, clk
    init update
      $[\!]\ clk' \rightarrow out' := in'$

module *IllegalLoop* =
  —**external** clk
  —**interface** $latch_1, latch_2$
  $\|\ UselessTransLatch[in, out, clk := latch_1, latch_2, clk]$
  $\|\ UselessTransLatch[in, out, clk := latch_2, latch_1, not\_clk]$
  $\|\ Not[in, out := clk, not\_clk]$

Figure 9: Naive model of a transparent latch

collapses consecutive rounds of P until one of the variables in Y changes its value. This is similar to the notion of sampling simulation for a complex system: we want to observe the behavior of the module P only at those instances when the value of some of the variables in Y changes. As we compress several rounds into one, it is assumed that an external variable that is read stays unchanged in all, except possibly the last, rounds, and an external variable that is awaited stays unchanged in all, except possibly the first, rounds.

Let P be a module, and let $Y \subseteq \mathit{intfX}_P$ be a subset of its interface variables. For two states s and t of P, the state t is a *Y-successor* of s if there exists a finite sequence $s_0 \ldots s_n$ of states of P such that the following three conditions are met:

1. $s_0 = s$; for all $0 \leq i < n$, the state s_{i+1} is a successor of s_i; and $s_n = t$.
2. For all $0 < i < n$, we have $s_i[Y] = s_0[Y]$; and $s_n[Y] \neq s_0[Y]$.
3. For every external variable x of P, if some atom of P reads x, then for all $0 < i < n$, we have $s_i[x] = s_0[x]$; and if some atom of P awaits x, then for all $0 < i < n$, we have $s_i[x] = s_n[x]$.

A *round marker* for the module P is a (nonempty) set Y of interface variables of P such that for every reachable state s of P, and every valuation t^e for the external variables of P, there is a nonzero and finite number of Y-successors t of s with $t[\mathit{extlX}_P] = t^e$. If Y is a round marker for P, then from any reachable state, no matter how the environment updates the external variables, 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, all of whose variables range over finite types, it can be checked automatically if Y is a round marker for P, by model checking a CTL formula of the form $\forall\Box\exists\mathcal{U}$.

Definition 8 [**Abstraction**] *Given a module P, if Y is a round marker for P, then the* abstraction **next** Y **for** P *is the 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 $\mathit{readX}_P = (\cup_{A \in \mathcal{A}_P} \mathit{readX}_A)$ of read variables, and the set $\mathit{waitX}_P = (\cup_{A \in \mathcal{A}_P} \mathit{waitX}_A) \cap \mathit{extlX}_P$ of awaited variables. The initial action of A_P^Y contains all pairs of the form $(s'[\mathit{waitX}'_P], s'[\mathit{ctrX}'_P])$, where s is an initial state of P. The update action of A_P^Y contains all pairs of the form $(s[\mathit{readX}_P] \cup t'[\mathit{waitX}'_P], t'[\mathit{ctrX}'_P])$, where t is a Y-successor of s.*

As an example, consider the two modules shown in Figure 10. The private variable *count* of the module P is initially 0. As long as the latched value of the external variable x is true, the variable *count* is incremented modulo 10. The interface event y is issued whenever *count* is incremented from 9 to 0. The interface event z is issued whenever x is false. The set $\{y, z\}$ is a round marker for P: every state with $x = \mathit{true}$ (and $\mathit{count} = i$) has a $\{y\}$-successor with $x = \mathit{true}$ (and $\mathit{count} = 0$) and a $\{y\}$-successor with $x = \mathit{false}$ (and $\mathit{count} = 0$); and every state with $x = \mathit{false}$ (and $\mathit{count} = i$) has a $\{z\}$-successor with $x = \mathit{true}$ (and $\mathit{count} = i$) and a $\{z\}$-successor with $x = \mathit{false}$ (and $\mathit{count} = i$). By contrast, neither $\{y\}$ nor $\{z\}$ are round markers for P, because the initial state with $x = \mathit{true}$ (and $\mathit{count} = 0$) does not have a $\{z\}$-successor, and the initial state with $x = \mathit{false}$ (and $\mathit{count} = 0$) does not have a $\{y\}$-successor. The abstraction **next** $\{y, z\}$ **for** P is $\cong$-equivalent to the module Q: whenever x is true, Q issues the interface event y, and whenever x is false, Q issues the interface event z.

It is easy to check that if Y is a round marker for a module 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

module P
  external $x : \mathbb{B}$
  interface $y, z : \mathbb{E}$
  private $count : [0..9]$
  atom $count$ **reads** $count, x$
    init
      $[\!] \ true \ \rightarrow \ count' := 0$
    update
      $[\!] \ x \wedge count = 0 \ \rightarrow \ count' := count + 1$
      $[\!] \ x \wedge 0 < count < 9 \ \rightarrow \ count' := count + 1$
      $[\!] \ x \wedge count = 9 \ \rightarrow \ count' := 0$
  atom y, z **reads** x, y, z **awaits** $count$
    update
      $[\!] \ count' = 0 \ \rightarrow \ y!$
      $[\!] \ \neg x \ \rightarrow \ z!$

module Q
  external $x : \mathbb{B}$
  interface $y, z : \mathbb{E}$
  atom y, z **reads** x, y, z
    update
      $[\!] \ x \ \rightarrow \ y!$
      $[\!] \ \neg x \ \rightarrow \ z!$

Figure 10: Round abstraction

marker for P. If the set $intfX_P$ of all interface variables is a round marker for P, then 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.

We now show that the abstraction operation is compositional. For this purpose, we need to strengthen the definition of the implementation relation slightly. The module P *environment-faithfully implements* the module Q, written $P \preceq^e Q$, if $P \preceq Q$ and for every external variable x of Q, (A) if x is read by some atom of Q, then x is an external variable of P that is read by some atom of P, and (B) if x is awaited by some atom of Q, then x is an external variable of P that is awaited by some atom of P. Environment-faithful implementations cannot constrain external variables by turning them into interface variables.

Proposition 7 *Let P and Q be two modules, and let Y be a round marker for both P and Q. Then $P \preceq^e Q$ implies* (**next** Y **for** P) $\preceq$ (**next** Y **for** Q).

Proof. Consider two modules $P' =$ (**next** Y **for** P) and $Q' =$ (**next** Y **for** Q). Assume that $P \preceq Q$, and assume conditions (A) and (B) of the definition for the environment-faithful implementation of Q by P. We prove that $P' \preceq Q'$. The definition of implementation has four conditions. The first two conditions are immediate. For the third condition, consider an observable variable x of Q' and an interface variable y of Q', suppose that $y \succ_{Q'} x$, and show that $y \succ_{P'} x$. Since $y \succ_{Q'} x$, and Q' has a single atom, x is an external variable of (Q' and) Q that is awaited by some atom of Q. From assumption (B), it follows that x is an external variable of P (and P') that is awaited by some atom of P. Since $P \preceq Q$, and y is an interface variable of (Q' and) Q, it is also an interface variable of P (and P'). Therefore $y \succ_{P'} x$.

The fourth condition is trace containment. Let $\overline{a}$ be a trace of P', and consider the trajectory $\overline{s} = s_0 \ldots s_n$ with $\overline{s}[obsX_{P'}] = \overline{a}$. Then, for all $0 \leq i < n$, the state s_{i+1} is a Y-successor of s_i according to P. Hence, by the definition of the **next** operator, we can introduce a finite sequence of states between each pair of states of $\overline{s}$ to obtain a trajectory of P of the form $s_0 s_{00} \ldots s_{0k_0} s_1 s_{10} \ldots s_{1k_1} s_2 \ldots s_n$. For each state s_{ij}, the value of the round marker Y equals $s_i[Y]$, the value of each read external variable x of P equals $s_i[x]$, and the value of each awaited external variable x of P equals $s_{i+1}[x]$. Since $P \preceq Q$, there exists a trajectory of Q of the form $t_0 t_{00} \ldots t_{0k_0} t_1 t_{10} \ldots t_{1k_1} t_2 \ldots t_n$ such that for all $0 \leq i \leq n$, we have $s_i[obsX_Q] = t_i[obsX_Q]$, and for all $0 \leq j \leq k_i$, we have $s_{ij}[obsX_Q] = t_{ij}[obsX_Q]$. From assumptions (A) and (B), it follows that for all $0 \leq i < n$, the state t_{i+1} is a Y-successor of t_i according to Q. Hence, $t_0 \ldots t_n$ is a trajectory of Q'. Therefore, $\overline{t}[obsX_{Q'}] = \overline{s}[obsX_{Q'}] = \overline{a}[obsX_{Q'}]$ is a trace of Q'. ■

From asynchrony to round-sensitive synchrony. Every module of the form **next** Y **for** P is synchronous and round-sensitive. Hence, abstraction is useful for constructing round-sensitive synchronous modules from asynchronous modules. For example, given the asynchronous counter *AsyncCount* from Figure 4, we can implement the round-sensitive synchronous counter *RoundCount* using abstraction:

RoundCount $\cong$ **next** *AsyncCount*

Similarly, while the message-passing implementation *SendRecImpl* from Section 5.3 is asynchronous, its stutter reduction

module *RedSendRec* $=$ **next** *SendRecImpl*

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.

module *Add64*
  external $x, y : \mathbb{B}[0..63]$; $carry : \mathbb{B}$
  interface $z : \mathbb{B}[0..63]$; $ofl : \mathbb{B}$
  atom z, ofl **awaits** $x, y, carry$
    init update
      $[\!]$ $true \rightarrow z' := (x' + y' + carry')$ **mod** 2^{64};
        $ofl' := (x' + y' + carry')$ **div** 2^{64}

module *ParAdd* $=$
  —**external** $x, y, carry$
  —**interface** z, ofl
  hide u **in**
    $\|$ $Add32[x, y, z, carry, ofl := x_0, y_0, z_0, carry, u]$
    $\|$ $Add32[x, y, z, carry, ofl := x_1, y_1, z_1, u, ofl]$

Figure 11: Specification and parallel implementation of a 64-bit adder

6.2 Temporal scaling

The **next** operator changes the notion of what happens during a round, and allows us to construct a time hierarchy of modules. This can again be illustrated with circuit examples. In a first example, we aggregate several clock cycles into an arithmetic operation; in a second example, we aggregate several gate operations into a clock cycle.

Consider the specification *Add64* of a 64-bit adder shown in Figure 11. 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 less significant 32 bits and x_1 for the more significant 32 bits. The first implementation, *ParAdd* (Figure 11), uses two copies of *Add32* and connects them appropriately. In each round, *ParAdd* adds two 64-bit words by first adding the less significant half-words and then, in a later subround of the same round, adding the more significant half-words. Hence $ParAdd \preceq Add64$.

The second implementation, *SeqAdd* (Figure 12), uses a single copy of *Add32* and embeds it in additional circuitry, represented by the module *AuxCircuitry*. The submodule

$$\textbf{module } SeqAddSub \;=\; Add32[x, y, z, carry, ofl := a, b, c, u, ofl] \;\|\; AuxCircuitry$$

requires two consecutive update rounds to compute a 64-bit sum: it adds the less significant half-words in one round before adding the more significant half-words in the subsequent round. The module *SeqAddSub* has four atoms. In each round, first the variable *round* is set to indicate if the less significant ($round = 0$) or the more significant ($round = 1$) half-words need to be added. Second, the variables a, b, and u are assigned the proper input values for the 32-bit addition. Third, the 32-bit addition is performed. Fourth, the output values of the 32-bit addition are assigned to the variables z and possibly v (intermediate carry), and if $round = 1$, the event *done* signals completion of the 64-bit addition. In *SeqAdd*, the two rounds of each 64-bit addition are collapsed, so that $SeqAdd \preceq Add64$. Indeed, all three models of the 64-bit adder are equivalent:

$$Add64 \;\cong\; ParAdd \;\cong\; SeqAdd$$

```
module SeqAdd =
  —external x, y, carry
  —interface z, ofl
  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]
    || AuxCircuitry

module AuxCircuitry
  external x, y: 𝔹[0..63]; c: 𝔹[0..31]; carry, ofl: 𝔹
  interface round: {0, 1}; z: 𝔹[0..63]; a, b: 𝔹[0..31]; u: 𝔹; done: 𝔼
  private v: 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 12: Sequential implementation of a 64-bit adder

```
module UsefulTransLatch
  external in, clk: 𝔹
  interface out: 𝔹
  atom out reads in, out awaits clk
    init update
      [] clk' → out' := in

module LegalLoop =
  —external clk
  —interface latch_1, latch_2
  stabilize hide not_clk in
    || UsefulTransLatch[in, out, clk := latch_1, latch_2, clk]
    || UsefulTransLatch[in, out, clk := latch_2, latch_1, not_clk]
    || Not[in, out := clk, not_clk]
```

Figure 13: Correct model of a transparent latch

Round abstraction is also useful for modeling systems that otherwise cannot be modeled naturally as reactive modules because of the acyclicity requirement on await dependencies. For example, using round abstraction, we can legally model the circuit *IllegalLoop* from Figure 9 using the scheme shown in Figure 13. Unlike the module *UselessTransLatch*, which awaits the data input *in*, the module *UsefulTransLatch* reads *in*, and thus delays its propagation to the output by a round. Then, for each constant control input *clk*, the variables of the module

module *LegalLoopSub* $=$
$\|$ $\mathit{UsefulTransLatch}[in, out, clk := latch_1, latch_2, clk]$
$\|$ $\mathit{UsefulTransLatch}[in, out, clk := latch_2, latch_1, not\_clk]$
$\|$ $\mathit{Not}[in, out := clk, not\_clk]$

stabilize within at finite number of rounds —that is, after some finite number of rounds, the variables of *LegalLoopSub* remain unchanged if any additional rounds are executed. When the variables stabilize, a fixpoint is reached for the values $latch_1$ and $latch_2$ of the transparent latches. This signals the end of a clock cycle. Then, the control input *clk* can change, and a new fixpoint iteration starts, whose result represents the state of the circuit after another clock cycle, etc.

Hence, we want to iterate the update actions of the module *LegalLoopSub* until the interface variables remain unchanged. This can be achieved by first composing *LegalLoopSub* with a module *WatchLegalLoopSub* that watches the execution of *LegalLoopSub* and issues the event *stable* once the interface variables of *LegalLoopSub* remain unchanged during an update round. In general, for an arbitrary module P, the monitor module *WatchP* is defined as follows:

module *WatchP*
  external intfX_P
  interface $\mathit{stable} : \mathbb{E}$
  atom *stable* **reads** $\mathit{stable}, \mathit{intfX}_P$ **awaits** intfX_P
    update
      $[]\ \mathit{intfX}_P' = \mathit{intfX}_P \ \rightarrow\ \mathit{stable}!$

Then we collapse rounds of *LegalLoopSub* $\|$ *WatchLegalLoopSub* until the event *stable* occurs. Let us write **stabilize** P as an abbreviation for the module **hide** *stable* **in next** $\{stable\}$ **for** $(P \parallel \mathit{WatchP})$. The result is the module *LegalLoop* of Figure 13. Within every context, the module *LegalLoop* properly updates the values of the transparent latches every single round.

Round abstraction in 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 $Q = (\textbf{next}\ Y\ \textbf{for}\ P)$. 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. For example, in Figure 10, in the abstract module **next** $\{y, z\}$ **for** P, the value of *count* is 0 in every reachable state. When G_Q is searched explicitly, 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 [14].

Also the symbolic verification of a system that consists of modules with **next** operators can be performed efficiently. Consider the module $P = (\textbf{next}\ Y_1\ \textbf{for}\ P_1) \parallel (\textbf{next}\ Y_2\ \textbf{for}\ P_2)$. Each single image-computation step for P corresponds to iterating the transition relation of P_1 until some variable in Y_1 changes, and iterating, independently, the transition relation of P_2 until some variable in Y_2 changes. The experiments reported in [5] indicate that this scheme can enable the analysis of P in cases where no analysis of $P_1\|P_2$ is feasible.

6.3 Triggering

Hiding allows us to build asynchronous modules from synchronous parts, and round abstraction allows us to build synchronous and round-sensitive modules from asynchronous and/or round-insensitive parts. We now introduce the operator **trigger** for building round-insensitive modules from round-sensitive parts. Intuitively, given a subset Z of the external variables of a module P, the module **trigger** Z **for** P sleeps until some external variable in Z changes its value. Then, P is executed. Thus, in a sense, triggering is dual to round abstraction: while the operator **next** collapses several rounds of a module into a single round, the operator **trigger** splits a round into several rounds, in all but one of which the module sleeps.

Let P be a module, and let $Z \subseteq extlX_P$ be a subset of its external variables. The set Z is a *read trigger* for P if $z \in Z$ for every external variable z of P that is read by some atom of P. The set Z is an *await trigger* for P if $z \in Z$ for every external variable z of P that is awaited by some atom of P. A *trigger* for P is either a read trigger or an await trigger.

Definition 9 [**Trigger**] *Given a module P, if Z is a trigger for P, then the module* **trigger** Z **for** P *has the same declaration as P and a single atom, B_P^Z. The atom B_P^Z has the set $ctrX_P$ of controlled variables, the set $readX_P^Z = (\cup_{A\in\mathcal{A}_P} readX_A) \cup Z$ of read variables, and the set $waitX_P^Z = ((\cup_{A\in\mathcal{A}_P} waitX_A) \cap extlX_P) \cup Z$ of awaited variables. The initial action of B_P^Z contains all pairs of the form $(s'[waitX'_P], s'[ctrX'_P])$, where s is an initial state of P. The update action of B_P^Z contains all pairs of the form $(s[readX_P] \cup t'[waitX'_P], t'[ctrX'_P])$, where either (1) $s[Z] = t[Z]$ and $s[ctrX_P] = t[ctrX_P]$, or (2) $s[Z] \neq t[Z]$ and t is a successor of s according to P.*

It is easy to check that if Z is a round marker for a module P, then the module **trigger** Z **for** P is again a module. Henceforth, whenever we write **trigger** Z **for** P, we assume that Z is a set of variables that contains no controlled variables of P. If Z contains some variables that are not (external) variables of P, then we agree that **trigger** Z **for** P stands for the module **trigger** Z **for** $(P\|Q)$, where Q is the trivial module with the set $Z\backslash extlX_P$ of external variables and the empty set of atoms. The 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 the update rounds in which the environment changes the value of some external variable, and sleeps in the update rounds in which the environment stutters.

Like the other operations on modules, triggering is compositional.

Proposition 8 *Let P and Q be two modules, and let Z be a trigger for both P and Q. Then $P \preceq^e Q$ implies* (**trigger** Z **for** P) $\preceq$ (**trigger** Z **for** Q).

Proof. Consider two modules $P' =$ (**trigger** Z **for** P) and $Q' =$ (**trigger** Z **for** Q). Assume that $P \preceq Q$, and assume conditions (A) and (B) of the definition for the environment-faithful implementation of Q by P. We prove that $P' \preceq Q'$. The definition of implementation has four conditions. The first two conditions are immediate, and the third condition can be shown by an argument similar to the one used in the proof of Proposition 7.

The fourth condition is trace containment. Let $\overline{a}$ be a trace of P', and consider the trajectory $\overline{s}$ with $\overline{s}[obsX_{P'}] = \overline{a}$. The trajectory $\overline{s}$ has form $s_{00} \ldots s_{0k_0} s_{10} \ldots s_{1k_1} s_{20} \ldots s_{nk_n}$ such that for all $0 \leq i < n$, we have $s_{ik_i}[Z] \neq s_{i+1,0}[Z]$, and $s_{i+1,0}$ is a successor of s_{ik_i} according to P, and for all $0 \leq i \leq n$ and $0 \leq j \leq k_i$, we have $s_{ij}[ctrX_P \cup Z] = s_{i0}[ctrX_P \cup Z]$. If Z is a read trigger for Q, then by assumption (A), the set Z is also a read trigger for P, and therefore $s_{00}s_{10} \ldots s_{n0}$ is a trajectory of P. If Z is an await trigger for Q, then by assumption (B), the set Z is also an await trigger for P, and therefore $s_{0k_0}s_{1k_1} \ldots s_{nk_n}$ is a trajectory of P. We pursue only the former case; the latter can be be handled similarly. Since $P \preceq Q$, there exists a trajectory of Q of the form $t_{00}t_{10} \ldots t_{n0}$ such that for all $0 \leq i \leq n$, we have $s_{i0}[obsX_Q] = t_{i0}[obsX_Q]$. Let $\overline{t}$ be a state sequence of the form $t_{00} \ldots t_{0k_0} t_{10} \ldots t_{1k_1} t_{20} \ldots t_{nk_n}$ such that for all $0 \leq i \leq n$ and $0 < j \leq k_i$, we have $t_{ij}[ctrX_Q \cup Z] = t_{i0}[ctrX_Q \cup Z]$ and $t_{ij}[extlX_Q \backslash Z] = s_{ij}[extlX_Q \backslash Z]$. Then, since Z is a read trigger for Q, the state sequence $\overline{t}$ is a trajectory of Q'. Therefore, $\overline{t}[obsX_{Q'}] = \overline{s}[obsX_{Q'}] = \overline{a}[obsX_{Q'}]$ is a trace of Q'. ■

From round-sensitivity to round-insensitivity. While triggering preserves (in spirit) asynchrony as well as synchrony, every module of the form **trigger** Z **for** P is round-insensitive. Hence, triggering is useful for constructing round-insensitive modules from round-sensitive modules. For example, given the round-sensitive synchronous counter *RoundCount* from Figure 4, we can implement the round-insensitive synchronous counter *EventCount* using triggering:

$$EventCount \;\cong\; \textbf{trigger}\; RoundCount$$

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

$$RoundCount \;\cong\; \textbf{next}\; AsyncCount$$

$$AsyncCount \;\cong\; \textbf{hide}\; tick\; \textbf{in}\; (EventCount \parallel Clock)$$

Reactive modules vs. multiform time. The temporal operators **next** and **trigger** of reactive modules are similar in spirit to the polychronous operators of synchronous programming languages such as SIGNAL [7] and LUSTRE [14]. Both approaches allow temporal abstraction by manipulating what happens during a round. However, there is a key difference. Reactive modules have a global notion of round, and applications of **next** and **trigger** only change what a module does within a round. In SIGNAL, the notion of round (or *clock*) is local to a module (or *signal*), is part of its semantics, and can be changed by applying operators such as **when** and **default**. Consequently, the parallel composition of reactive modules behaves quite differently from the parallel composition in synchronous programming languages.

7 Fair Reactive Modules

Based on the trace semantics of modules from Section 4, we can reason about the safety requirements of modules. Reasoning about liveness requirements demands that we consider *infinite* behaviors of modules. Then, in order to rule out certain degenerate infinite behaviors of a module, we add fairness constraints to the module.

7.1 The infinite traces of a module

Let P be a module. An *ω-trajectory* of P is an infinite sequence $s_0s_1s_2\ldots$ of states such that (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. If $\underline{s} = s_0s_1\ldots$ is an

ω-trajectory of P, then the corresponding infinite sequence $\underline{s}[obsX_P] = s_0[obsX_P]s_1[obsX_P]\ldots$ of observations is an *ω-trace* of P. Since all initial and update actions of the module P are executable, the set of ω-traces of P is completely determined by the set L_P of finite traces, and vice versa. This property of a reactive system is called *limit closure*, or *safety* [3].

Proposition 9 *Let P be a module. An infinite sequence $\underline{a}$ of observations of P is an ω-trace of P iff every finite prefix of $\underline{a}$ is a trace of P. A finite sequence $\overline{a}$ of observations of P is a trace of P iff $\overline{a}$ is a finite prefix of some ω-trace of P.*

Proof. The first part of the proposition follows from the finite controlled branching of the transition relation. Consider an infinite sequence $\underline{a} = a_0 a_1 \ldots$ of observations. If $\underline{a}$ is an ω-trace of P, then, by definition, every finite prefix of $\underline{a}$ is a trace of P. So suppose that for all $i \geq 0$, the finite sequence $\overline{a}^i = a_0 \ldots a_i$ is a trace of P; that is, for all $i \geq 0$, there is a finite trajectory $\overline{s}^i$ of P with $\overline{s}^i[obsX_P] = \overline{a}^i$. We define a forest whose vertices are labeled with states of P. For every initial s of P with $s[obsX_P] = a_0$, there is a root —i.e., a level-0 vertex— labeled with s. For every level $i \geq 0$, every level-i vertex v labeled with state s, and every successor t of s with $t[obsX_P] = a_{i+1}$, there is a child of v —i.e., a level-$(i+1)$ vertex— labeled with t. Since all initial and update actions of P are executable, the forest has a finite number roots, and every vertex has a finite number of children. Furthermore, for each $i \geq 0$, the finite trajectory $\overline{s}^i$ is a path of the forest. Hence the forest has infinitely many vertices, and by König's lemma, the forest contains an infinite path. The sequence of labels along this path is an ω-trajectory of P, and the corresponding ω-trace is $\underline{a}$.

The second part of the proposition follows from the seriality of the transition relation. ■

7.2 Modules with fairness constraints

Remember that an action α from X to Y is a binary relation between the valuations for X and the valuations for Y. Thus, all subsets $\beta \subseteq \alpha$ are also actions from X to Y; they are called the *subactions* of α. An *update choice* for an atom A is a subaction of the update action $Update_A$. An update choice need not be executable; that is, an update choice may be enabled in some states but not in others. We add fairness constraints to modules by declaring a set of weakly-fair update choices and a set of strongly-fair update choices for each atom.

Definition 10 [**Fair modules**] *A* fair module *$\mathcal{P}$ consists of a module $safe(\mathcal{P})$ together with two fairness constraints. The* weak-fairness constraint *$wf_{\mathcal{P}}$ is a function that maps every atom A of $safe(\mathcal{P})$ to a finite set of update choices for A, which are called the* weakly-fair *update choices for A. The* strong-fairness constraint *$sf_{\mathcal{P}}$ is a function that maps every atom A of $safe(\mathcal{P})$ to a finite set of update choices for A, which are called the* strongly-fair *update choices for A.*

For a fair module $\mathcal{P}$, we refer to parts of the underlying module $safe(\mathcal{P})$ —such as variables, atoms, etc.— as parts of $\mathcal{P}$. The fair module $\mathcal{P}$ is *weakly fair* if for every atom A of $\mathcal{P}$, the set $sf_{\mathcal{P}}(A)$ of strongly-fair update choices is empty. The fair module $\mathcal{P}$ is *trivially fair* if for every atom A of $\mathcal{P}$, both sets $wf_{\mathcal{P}}(A)$ and $sf_{\mathcal{P}}(A)$ of update choices are empty.

The fairness constraints of the fair module $\mathcal{P}$ classify the ω-trajectories of the underlying module $safe(\mathcal{P})$ into fair and unfair. Intuitively, a weakly-fair update choice cannot be enabled forever without being chosen, and a strongly-fair update choice cannot be enabled infinitely often without being chosen.

module *Fair*
  external $x : \mathbb{E}$
  interface $y, z : \mathbb{E}$
  atom y, z **reads** x, y, z **awaits** x
    update weakly-fair α **strongly-fair** β
      $[\!]\ x? \xrightarrow{\alpha} y!$
      $[\!]\ x? \xrightarrow{\beta} z!$

Figure 14: Weak and strong fairness

Consider an update choice α for the atom A, and an infinite sequence $\underline{s}$ of states. The update choice α is *enabled* at position $i \geq 0$ of $\underline{s}$ if there is a state t such that $(s_i[readX_A] \cup s'_{i+1}[waitX'_A], t'[ctrX'_A]) \in \alpha$. The update choice α is *chosen* at position $i \geq 0$ of $\underline{s}$ if $(s_i[readX_A] \cup s'_{i+1}[waitX'_A], s'_{i+1}[ctrX'_A]) \in \alpha$. The state sequence $\underline{s}$ is *weakly fair* to the update choice α if either α is not enabled at infinitely many positions of $\underline{s}$, or α is chosen at infinitely many positions of $\underline{s}$. The state sequence $\underline{s}$ is *strongly fair* to the update choice α if either α is enabled at only finitely many positions of $\underline{s}$, or α is chosen at infinitely many positions of $\underline{s}$. A *fair trajectory* of the fair module $\mathcal{P}$ is an ω-trajectory $\underline{s}$ of the module *safe*$(\mathcal{P})$ such that for every atom A of $\mathcal{P}$, the state sequence $\underline{s}$ is weakly fair to all update choices in $wf_{\mathcal{P}}(A)$ and strongly fair to all update choices in $sf_{\mathcal{P}}(A)$. If $\underline{s}$ is a fair trajectory of $\mathcal{P}$, then the corresponding infinite sequence $\underline{s}[obsX_{\mathcal{P}}]$ of observations is a *fair trace* of $\mathcal{P}$.

Example. Consider the fair module *Fair* shown in Figure 14. In every update round, if the external event x is present, then the module issues, nondeterministically, either the interface event y or the interface event z. The update choice α consists of all pairs (s, t) of states such that $s[x] \neq t[x]$ and $s[y] \neq t[y]$, and the update choice β consists of all pairs (s, t) such that $s[x] \neq t[x]$ and $s[z] \neq t[z]$. The weak-fairness assumption for α ensures that in every fair trace, if, after some round, x is present in every round, then y is issued infinitely often. The strong-fairness assumption for β ensures that in every fair trace, if x is present in infinitely many rounds, then z is issued infinitely often. $\square$

The fairness constraints of a fair module can be translated into ω-acceptance conditions on the underlying state-transition graph —weak-fairness constraints into Büchi conditions, and strong-fairness constraints into Streett conditions. However, while ω-acceptance conditions are usually defined using sets of states, a direct translation of the fairness constraints on modules leads to ω-acceptance conditions that are defined using sets of transitions. This is because whether an update choice is enabled or chosen may depend on both the latched and the updated values of variables.

Definition 11 [**Fair implementation**] *The fair module $\mathcal{P}$* fairly implements *the fair module $\mathcal{Q}$, written $\mathcal{P} \preceq^F \mathcal{Q}$, if the first three conditions of Definition 3 are met, and (4) if $\underline{s}$ is a fair trace of $\mathcal{P}$, then the projection $\underline{s}[obsX_{\mathcal{Q}}]$ is a fair trace of $\mathcal{Q}$.*

It is easy to check that the fair-implementation relation $\preceq^F$ is a preorder on fair modules.

Machine closure. Every finite trajectory of a fair module $\mathcal{P}$ can be extended to a fair trajectory of $\mathcal{P}$. This property of a reactive system is called *machine closure* [1].

Proposition 10 *If $\mathcal{P}$ is a fair module, and $\overline{a}$ is a finite trace of $\mathcal{P}$, then $\overline{a}$ is a finite prefix of some fair trace of $\mathcal{P}$.*

It follows that the set of fair trajectories of a fair module is always nonempty. Moreover, in verification, machine closure is important for two reasons. First, as we will see in the next section, machine closure facilitates an assume-guarantee principle for fair implementation. Second, the fairness constraints of a machine-closed system can be ignored when reasoning about safety requirements of the system. Both are because for machine-closed systems, fair implementation implies implementation.

Proposition 11 *Let $\mathcal{P}$ and $\mathcal{Q}$ be two fair modules. Then $\mathcal{P} \preceq^F \mathcal{Q}$ implies $safe(\mathcal{P}) \preceq safe(\mathcal{Q})$. If $\mathcal{Q}$ is trivially fair, then $safe(\mathcal{P}) \preceq safe(\mathcal{Q})$ implies $\mathcal{P} \preceq^F \mathcal{Q}$.*

Proof. The first part of the proposition follows from the machine closure of fair modules (Proposition 10); the second part follows from the limit closure of trivially-fair modules (first part of Proposition 9). ■

Thus, in order to show that a fair module $\mathcal{P}$ fairly implements a trivially-fair module $\mathcal{Q}$, which represents a safety requirement of $\mathcal{P}$, it is sufficient and necessary to show that $safe(\mathcal{P})$ implements $safe(\mathcal{Q})$.

Receptiveness. Proposition 10 can be strengthened: in order to extend a finite trajectory to a fair trajectory, the module does not need the cooperation of the environment. Given a fair module $\mathcal{P}$, consider a finite trajectory $s_0 \ldots s_n$ of the underlying module $safe(\mathcal{P})$, and an infinite sequence $t^e_{n+1} t^e_{n+2} t^e_{n+3} \ldots$ of valuations for the external variables of $\mathcal{P}$. Then, there is a fair trajectory $\underline{u}$ of $\mathcal{P}$ such that for all $0 \leq i \leq n$, we have $u_i = s_i$, and for all $i > n$, we have $u_i[extlX_{\mathcal{P}}] = t_i$. In fact, even in a stepwise game between module and environment, no matter how the environment plays, the module always has a strategy to produce a fair trajectory. This property of a reactive system is called *receptiveness* [11]. In a formalism that builds compound systems from atomic systems, in order to prove that all compound systems are machine-closed, it suffices to prove that all atomic systems are receptive. Since fair modules will be closed under composition —that is, the parallel composition of two fair modules is again a fair module— there is no need to formally define and establish the receptiveness of fair modules.

7.3 Operations on fair modules

The operations of renaming, composition, and hiding extend to fair modules in the obvious way. For a fair module $\mathcal{P}$, and two variables x and y of the same type with $y \notin X_{\mathcal{P}}$, the fair module $\mathcal{P}[x := y]$ results from $\mathcal{P}$ by renaming x to y. The requirement of compatibility for fair modules is the same as for unfair modules. For two compatible fair modules $\mathcal{P}$ and $\mathcal{Q}$, $safe(\mathcal{P} \| \mathcal{Q}) = safe(\mathcal{P}) \parallel safe(\mathcal{Q})$, for every atom A of $\mathcal{P}$, we have $wf_{\mathcal{P}\|\mathcal{Q}}(A) = wf_{\mathcal{P}}(A)$ and $sf_{\mathcal{P}\|\mathcal{Q}}(A) = sf_{\mathcal{P}}(A)$, and for every atom A of $\mathcal{Q}$, we have $wf_{\mathcal{P}\|\mathcal{Q}}(A) = wf_{\mathcal{Q}}(A)$ and $sf_{\mathcal{P}\|\mathcal{Q}}(A) = sf_{\mathcal{Q}}(A)$. For a fair module $\mathcal{P}$ and a variable x, the fair module **hide** x **in** $\mathcal{P}$ results by moving x from $intfX_{\mathcal{P}}$ to $privX_{\mathcal{P}}$.

Example. In asynchronous shared-memory programs, progress can be ensured by weak-fairness constraints. Recall Peterson's solution to the mutual-exclusion problem from Figure 2. Figure 15 adds weak-fairness assumptions to the first process of the protocol; the second fair process is defined similarly, by the weakly-fair module $FairP_2$. The weak-fairness assumption for, say, the update choice β ensures that in every fair trace, it cannot happen that the first process remains in its critical section forever. For the unfair module $P_1 \| P_2$, we can prove mutual exclusion: in every trace, it cannot happen that both processes are simultaneously in their respective critical sections. For the weakly-fair module $FairP_1 \parallel FairP_2$, we can, in addition, prove starvation freedom: in every fair trace, if a process requests to enter its critical section, then eventually it will be in its critical section (some unfair ω-traces do not satisfy this requirement). □

module $FairP_1$
  interface $pc_1 : \{outCS, reqCS, inCS\};\ x_1 : \mathbb{B}$
  external $pc_2 : \{outCS, reqCS, inCS\};\ x_2 : \mathbb{B}$
  atom pc_1, x_1 **reads** pc_1, pc_2, x_1, x_2
    init
      $[\!]\ true\ \rightarrow\ pc_1' := outCS$
    update weakly-fair α, β
      $[\!]\ pc_1 = outCS\ \rightarrow\ pc_1' := reqCS;\ x_1' := x_2$
      $[\!]\ pc_1 = reqCS \wedge (pc_2 = outCS \vee x_1 \neq x_2)\ \xrightarrow{\alpha}\ pc_1' := inCS$
      $[\!]\ pc_1 = inCS\ \xrightarrow{\beta}\ pc_1' := outCS$
      $[\!]\ true\ \rightarrow$

Figure 15: Fair mutual-exclusion protocol

module $\mathcal{P}_1$
  external $y : \mathbb{B}$
  interface $x : \mathbb{B}$
  atom x **reads** y
    update
      $[\!]\ true\ \rightarrow\ x' := y$

module $\mathcal{P}_2$
  external $x : \mathbb{B}$
  interface $y : \mathbb{B}$
  atom y **reads** x
    update
      $[\!]\ true\ \rightarrow\ y' := x$

module $\mathcal{Q}_1$
  interface $x : \mathbb{B}$
  atom x
    update weakly-fair α
      $[\!]\ true\ \rightarrow\ x' := 0$
      $[\!]\ true\ \xrightarrow{\alpha}\ x' := 1$

module $\mathcal{Q}_2$
  interface $y : \mathbb{B}$
  atom y
    update weakly-fair β
      $[\!]\ true\ \rightarrow\ y' := 0$
      $[\!]\ true\ \xrightarrow{\beta}\ y' := 1$

Figure 16: Counterexample to naive assume-guarantee principle for fair modules

The three operations of renaming, composition, and hiding for fair modules are compositional with respect to fair implementation. For parallel composition, this follows from the analogue of Proposition 3: for two compatible fair modules $\mathcal{P}$ and $\mathcal{Q}$, an infinite sequence $\underline{a}$ of observations of the compound module $\mathcal{P}\|\mathcal{Q}$ is a fair trace of $\mathcal{P}\|\mathcal{Q}$ iff the projection $\underline{a}[obsX_{\mathcal{P}}]$ is a fair trace of $\mathcal{P}$ and the projection $\underline{a}[obsX_{\mathcal{Q}}]$ is a fair trace of $\mathcal{Q}$.

Proposition 12 *Let $\mathcal{P}$, $\mathcal{Q}$, and $\mathcal{R}$ be three fair modules such that $\mathcal{P}$ and $\mathcal{R}$ are compatible. Then (1) $\mathcal{P}\|\mathcal{R} \preceq^F \mathcal{P}$, and (2) $\mathcal{P} \preceq^F \mathcal{Q}$ implies $\mathcal{P}\|\mathcal{R} \preceq^F \mathcal{Q}\|\mathcal{R}$.*

Proposition 13 *For all fair modules $\mathcal{P}$ and $\mathcal{Q}$, and every variable x, (1) $\mathcal{P} \preceq^F$ (**hide** x **in** $\mathcal{P}$), and (2) if $\mathcal{P} \preceq^F \mathcal{Q}$, then (**hide** x **in** $\mathcal{P}$) $\preceq^F$ (**hide** x **in** $\mathcal{Q}$).*

Assume-guarantee reasoning. Recall the assume-guarantee principle for unfair modules from Proposition 5. Suppose that both $\mathcal{P}_1\|\mathcal{Q}_2 \preceq^F \mathcal{Q}_1$ and $\mathcal{Q}_1\|\mathcal{P}_2 \preceq^F \mathcal{Q}_2$. From this, we can conclude that every ω-trace of $safe(\mathcal{P}_1)\|safe(\mathcal{P}_2)$ is an ω-trace of $safe(\mathcal{Q}_1)\|safe(\mathcal{Q}_2)$. However, we cannot conclude that every fair trace of $\mathcal{P}_1\|\mathcal{P}_2$ is a fair trace of $\mathcal{Q}_1\|\mathcal{Q}_2$. Figure 16 shows a counterexample, for the special case that both $\mathcal{P}_1$ and $\mathcal{P}_2$ are trivially fair: while $\mathcal{P}_1\|\mathcal{Q}_2$ guarantees that infinitely often $x = 1$, and $\mathcal{Q}_1\|\mathcal{P}_2$ guarantees that infinitely often $y = 1$, the module $\mathcal{P}_1\|\mathcal{P}_2$ guarantees neither. The circularity in the fairness constraints needs to be broken, which leads to a somewhat weaker form of assume-guarantee principle in the presence of fairness [4]. For a fair module $\mathcal{P}$, let $unfair(\mathcal{P})$ be the trivially-fair module $\mathcal{Q}$ with $safe(\mathcal{Q}) = safe(\mathcal{P})$; that is, $unfair(\mathcal{P})$ is obtained from $\mathcal{P}$ by discarding the fairness constraints.

Proposition 14 *Let $\mathcal{P}_1$ and $\mathcal{P}_2$ be two compatible fair modules, and let $\mathcal{Q}_1$ and $\mathcal{Q}_2$ be two compatible fair modules such that every external variable of $\mathcal{Q}_1\|\mathcal{Q}_2$ is an observable variable of $\mathcal{P}_1\|\mathcal{P}_2$. If $\mathcal{P}_1\|\mathcal{Q}_2 \preceq^F \mathcal{Q}_1$ and $unfair(\mathcal{Q}_1)\|\mathcal{P}_2 \preceq^F \mathcal{Q}_2$, then $\mathcal{P}_1\|\mathcal{P}_2 \preceq^F \mathcal{Q}_1\|\mathcal{Q}_2$.*

Proof. Consider four fair modules $\mathcal{P}_1$, $\mathcal{P}_2$, $\mathcal{Q}_1$, and $\mathcal{Q}_2$ such that (1) $\mathcal{P}_1$ and $\mathcal{P}_2$ are compatible, (2) $\mathcal{Q}_1$ and $\mathcal{Q}_2$ are compatible, (3) every external variable of $\mathcal{Q}_1\|\mathcal{Q}_2$ is an observable variable of $\mathcal{P}_1\|\mathcal{P}_2$, (4) $\mathcal{P}_1\|\mathcal{Q}_2 \preceq^F \mathcal{Q}_1$, and (5) $unfair(\mathcal{Q}_1)\|\mathcal{P}_2 \preceq^F \mathcal{Q}_2$. We wish to establish that $\mathcal{P}_1\|\mathcal{P}_2 \preceq^F \mathcal{Q}_1\|\mathcal{Q}_2$. From assumptions (4) and (5), by the first part of Proposition 11 it follows that $safe(\mathcal{P}_1)\|safe(\mathcal{Q}_2) \preceq safe(\mathcal{Q}_1)$ and $safe(\mathcal{Q}_1)\|safe(\mathcal{P}_2) \preceq safe(\mathcal{Q}_2)$. From this and assumptions (1)–(3), by the assume-guarantee principle for unfair modules (Proposition 5) it follows that $safe(\mathcal{P}_1)\|safe(\mathcal{P}_2) \preceq safe(\mathcal{Q}_1)\|safe(\mathcal{Q}_2)$. This implies the first three conditions of the definition of fair implementation. Hence it remains to be shown that every fair trace of $\mathcal{P}_1\|\mathcal{P}_2$ is also a fair trace of $\mathcal{Q}_1\|\mathcal{Q}_2$ (for simplicity, we omit the explicit use of projections).

Let $\underline{a}$ be a fair trace of $\mathcal{P}_1\|\mathcal{P}_2$, and therefore of both $\mathcal{P}_1$ and $\mathcal{P}_2$. Since every trace of $safe(\mathcal{P}_1)\|safe(\mathcal{P}_2)$ is a trace of $safe(\mathcal{Q}_1)\|safe(\mathcal{Q}_2)$, by the first part of Proposition 9 it follows that $\underline{a}$ is an ω-trace of $safe(\mathcal{Q}_1)\|safe(\mathcal{Q}_2)$, and therefore of $safe(\mathcal{Q}_1)$. For a trivially-fair module $\mathcal{R}$, the fair traces coincide with the ω-traces of the underlying module $safe(\mathcal{R})$. Hence, $\underline{a}$ is a fair trace of $unfair(\mathcal{Q}_1)$. Since $\underline{a}$ is also a fair trace of $\mathcal{P}_2$, it follows that $\underline{a}$ is a fair trace of $unfair(\mathcal{Q}_1)\|\mathcal{P}_2$, and by assumption (5), a fair trace of $\mathcal{Q}_2$. Since $\underline{a}$ is also a fair trace of $\mathcal{P}_1$, it follows that $\underline{a}$ is a fair trace of $\mathcal{P}_1\|\mathcal{Q}_2$, and by assumption (4), a fair trace of $\mathcal{Q}_1$. Since $\underline{a}$ is a fair trace of both $\mathcal{Q}_1$ and $\mathcal{Q}_1$, we conclude that $\underline{a}$ is a fair trace of $\mathcal{Q}_1\|\mathcal{Q}_2$. ■

Round abstraction vs. fairness. The abstraction operator **next** is closely related to weak fairness. For instance, while not all ω-traces of the module $P_1\|P_2$ from Figure 2 satisfy starvation freedom, all ω-traces of the stutter reduction **next** $(P_1\|P_2)$ do. Indeed, the module **next** $(P_1\|P_2)$ satisfies the stronger requirement of *bounded* starvation freedom: if a process requests to enter its critical section, then it will be in its critical section within four rounds.

8 Concluding Remarks

We have presented a unified, modular, and hierarchical framework for describing synchronous and asynchronous reactive computation. The uniformity, modularity, and hierarchy of reactive modules can be exploited in 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 progress, and BDD-based model checking is successful in this domain. On the other hand, many protocol description languages (like PROMELA [17]) assume asynchronous interleaving, and the most effective verification strategy is explicit on-the-fly search with reduction techniques based on partial orders and symmetries. Finally, the verification tools for synchronous programming languages (like ESTEREL [8]) can afford to construct global state-transition graphs, because much of the complexity is hidden by the fact that a single transition involves several subtransitions 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 [19]. Or, the introduction of synchronization points into asynchronous models restricts the applicability of efficient search methods in verification [17]. By contrast, our uniform framework allows us to separate intrinsic truths and complexities about verification methods from accidental and model-dependent idiosyncrasies.

In addition, our framework supports modular proof principles, such as assume-guarantee reasoning, and hierarchical verification, based on built-in abstraction operators such as **next**. This allows us to decompose a verification task into subtasks with smaller state spaces. Module-based case studies that exploit assume-guarantee reasoning can be found in [15]; case studies that expoit round abstraction, in [5]. A verification tool, called MOCHA, whose system description language is based on reactive modules, is currently being implemented [6].

Acknowledgments. We thank Albert Benveniste, Bob Kurshan, Ken McMillan, Amir Pnueli, and the VIS group at UC Berkeley for fruitful discussions. We also thank the anonymous referees for suggesting improvements.

References

[1] M. Abadi and L. Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82:253–284, 1991.

[2] M. Abadi and L. Lamport. Conjoining specifications. *ACM Transactions on Programming Languages and Systems*, 17:507–534, 1995.

[3] B. Alpern, A.J. Demers, and F.B. Schneider. Safety without stuttering. *Information Processing Letters*, 23:177–180, 1986.

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

[5] R. Alur, T.A. Henzinger, and S.K. Rajamani. Symbolic exploration of transition hierarchies. In *TACAS 98: Tools and Algorithms for the Construction and Analysis of Systems*, Lecture Notes in Computer Science 1384, pages 330–344, Springer-Verlag, 1998.

[6] R. Alur, T.A. Henzinger, F.Y.C. Mang, S. Qadeer, S.K. Rajamani, and S. Tasiran. MOCHA: modularity in model checking. In *CAV 98: Computer-aided Verification*, Lecture Notes in Computer Science, to appear. Springer-Verlag, 1998.

[7] A. Benveniste, P. le Guernic, and C. Jacquemot. Synchronous programming with events and relations: the SIGNAL language and its semantics. *Science of Computer Programming*, 16:103–149, 1991.

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

[9] 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.

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

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

[12] D.L. Dill. The MURϕ verification system. In *CAV 96: Computer-aided Verification*, Lecture Notes in Computer Science 1102, pages 390–393. Springer-Verlag, 1996.

[13] O. Grumberg and D.E. Long. Model checking and modular verification. *ACM Transactions on Programming Languages and Systems*, 16:843–871, 1994.

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

[15] T.A. Henzinger, S. Qadeer, and S.K. Rajamani. You assume, we guarantee: methodology and case studies. In *CAV 98: Computer-aided Verification*, Lecture Notes in Computer Science, to appear. Springer-Verlag, 1998.

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

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

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

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

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

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

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

[23] K.L. McMillan. *Symbolic Model Checking: An Approach to the State-explosion Problem.* Kluwer Academic Publishers, 1993.

[24] R. Milner. *Communication and Concurrency.* Prentice-Hall, 1989.

[25] E.W. Stark. A proof technique for rely-guarantee properties. In *FST & TCS 85: Foundations of Software Technology and Theoretical Computer Science*, Lecture Notes in Computer Science 206, pages 369–391. Springer-Verlag, 1985.

[26] R.J. van Glabbeek. *Comparative Concurrency Semantics and Refinement of Actions*. PhD thesis, Vrije Universiteit te Amsterdam, 1990.