

Asynchronous Progress

Ernie Cohen

August 30, 2004

Abstract

We propose weakening the definition of progress to a branching-time operator, making it more amenable to compositional proof and simplifying the predicates needed to reason about highly asynchronous programs. The new progress operator (“achieves”) coincides with the “leads-to” operator on all “observable” progress properties (those where the target predicate is stable) and satisfies the same composition properties as leads-to, including the PSP theorem. The advantage of achievement lies in its compositionality: a program inherits all achievement properties of its “decoupled components”. (For example, a dataflow network inherits achievement properties from each of its processes.) The compositionality of achievement captures, in a UNITY-like logic, the well-known operational trick of reasoning about an asynchronous program by considering only certain well-behaved executions.

1 Introduction

It is well-known that progress properties (such as leads-to [1]) are not preserved under parallel composition. That is, it is not generally possible to obtain a useful progress property of a system from a progress property of one of its components. Instead, it is usually necessary to work globally with atomic progress steps (obtained by combining local liveness properties with global safety properties). This often results in unreasonably complicated proofs.

For example, consider the following trivial producer-consumer system. The producer repeatedly chooses a function, applies it to his local value, and sends the function along a FIFO channel to the consumer; the consumer, on receiving a function, applies it to his local variable. Initially, the two variables are equal and the channel is empty; we call such a state *clean*. We would like to prove that, if the producer eventually stops, the system terminates in a clean state.

Assertional proofs of this program are rather painful; they generally require either the introduction of an auxiliary inductive definition that captures the behavior of one of the processes (e.g., in order to formulate an invariant like “the system is in a state reachable by running the producer from a clean state”) or introducing history variables on the channel (which amounts to reasoning

about a static execution object instead of a dynamic program). In either case, the programming logic fails to provide substantial reasoning leverage.

A more attractive, though informal, operational argument might go as follows: starting from a clean state, the producer is guaranteed to execute first. At this point, the producer cannot interfere with the consumer's first step, so we can pretend that the consumer executes next, bringing the system back to a clean state. This is repeated for every message sent by the producer, so when the system halts, the state is again clean.

There are a number of programming theorems that attempt to systematize this sort of reasoning, typically using commutativity to turn arbitrary execution fragments into well-behaved ones. However, all such approaches have to overcome a common barrier in order to be used with a linear-time programming logic: eventually, the technique has to reflect back to a concrete, reasonably simple linear-time property. For example, in the producer consumer example above, a reduction theorem could be used to reason about the entire execution, but not about a single step.

We propose a new progress operator \rightsquigarrow (“achieves”), that supports this kind of reasoning within the UNITY programming logic. Achievement has a number of attractive features:

- It supports the key reasoning rules of the UNITY leads-to operator; in particular, it is transitive, disjunctive, and satisfies the PSP theorem.
- It coincides with leads-to on those properties that are “observable” (i.e., those whose target predicates are stable). Thus, it is as expressive as leads-to for all practical purposes.
- Unlike leads-to, it supports a form of compositional reasoning: a program inherits achievement properties from each of its “decoupled components” (e.g., the processes of a dataflow network). Decoupling can itself be established compositionally, usually through simple structural analysis.
- Most techniques for reasoning about concurrency control (such as reduction [9, 4] or serializability [6]) are based on pretending that certain operations execute “atomically”. Decoupled components, on the other hand, effectively execute “immediately”. This makes them easier to compose and allows them to serve as asynchronous maintainers as invariants (section 6).
- Achievement and decoupling are defined semantically (i.e., in terms of the properties of a program, not its transitions). They are thus independent of program presentation, unlike related theories like communication-closed layers or stubborn sets [14], which are defined at the level of transitions.
- Unlike techniques for compositional temporal logic reasoning, our theory allows multiple processes to write to the same variable. This allows us to reason about FIFO channels without having to resort to history variables.

- Unlike interleaving set temporal logic [8], which requires reasoning about entire executions, achievement obtains the same effect using simpler UNITY-like program reasoning.

In this paper, we show the key concepts and theorems, and some simple examples. Proofs of all results can be found in [2], which also contains a number of examples, including the sliding window protocol and the tree protocol for database concurrency control.

2 Programs

Our programs are countable, unconditionally fair, nondeterministic transition systems. As a starting point, we describe them using operators from UNITY [1].

Let S be a fixed set of states. As usual, we describe subsets of S using state predicates (notation: p, q, r, s), and identify elements of S with their characteristic predicates. An *action* (notation: f, g) is a binary relation on S ; we identify actions with (universally disjunctive) predicate transformers giving their strongest postconditions. We will make use of the following actions (given in order of decreasing binding power):

$$\begin{aligned}
1.p &= p \\
0.p &= \text{false} \\
(f; g).p &= g.(f.p) \\
(\wedge q).p &= p \wedge q \\
(f \vee g).p &= f.p \vee g.p \\
(f \Rightarrow g).p &= f.p \Rightarrow g.p \\
(\exists x).p &= (\exists x : p) \\
x := e &= (\exists x'); (\wedge(x' = e)); (\exists x); (\wedge(x = x')); (\exists x') \\
&\quad \text{where } x' \text{ is a fresh variable} \\
(f \text{ if } p) &= ((\wedge p); f \vee (\wedge \neg p))
\end{aligned}$$

The everywhere operator of [5] is extended to predicate transformers by

$$[h] \equiv [(\forall p : h.p)]$$

A *program* (notation: A, B, \dots) is a countable set of actions. A program is executed by repeatedly stuttering or executing one of its actions, subject to the restriction that each action is chosen infinitely often; formally, an execution e is an infinite sequence of states e_i such that

$$\begin{aligned}
&(\forall i \geq 0 : (\exists a \in A \setminus \{1\} : [e_{i+1} \Rightarrow a.e_i])) \\
\wedge &(\forall i \geq 0, a \in A : (\exists j \geq i : [e_{j+1} \Rightarrow a.e_j]))
\end{aligned}$$

Under this semantics, union of programs corresponds to fair parallel composition, so we use the symbol $|$ as a synonym for set union when composing programs.

The motivating problem of this paper is the desire to prove properties of the form $(p \mapsto q \text{ in } A)$ (“ p leads-to q in A ”), which says that every execution of A that starts with a p -state contains a q -state:

$$(p \mapsto q \text{ in } A) \equiv (\forall e : e \text{ an execution of } A : [e_0 \Rightarrow p] \Rightarrow (\exists i : [e_i \Rightarrow q]))$$

The standard way to prove \mapsto properties is with the operators \mathbf{U} and \mathbf{E} , defined by

$$\begin{aligned} (p \mathbf{U} q \text{ in } A) &\equiv (\forall a \in A : (\wedge(p \wedge \neg q)); a; (\wedge(\neg p \wedge \neg q)) = 0) \\ (p \mathbf{E} q \text{ in } A) &\equiv (p \mathbf{U} q \text{ in } A) \wedge (\exists a \in A : (\wedge(p \wedge \neg q)); a; (\wedge \neg q) = 0) \end{aligned}$$

$(p \mathbf{U} q \text{ in } A)$ (“ p unless q in A ”) means that no A transition falsifies p without truthifying q (unless q is true already); this also means that in any execution of A , p , once true, remains true up to the first moment (if any) when q holds. $(p \mathbf{E} q \text{ in } A)$ (“ p ensures q in A ”) means that, in addition, some transition of A is guaranteed to yield a q state when executed from a $p \wedge \neg q$ state. \mathbf{U} properties are used to specify safety, while \mathbf{E} properties specify “atomic” progress.

Given \mathbf{U} and \mathbf{E} , we have the following (complete set of) rules for deriving \mapsto properties:

$$\begin{aligned} (p \mathbf{E} q) &\Rightarrow (p \mapsto q) & (1) \\ (p \mapsto q) \wedge (q \mapsto r) &\Rightarrow (p \mapsto r) \\ (\forall i : p_i \mapsto q_i) &\Rightarrow (\exists i : p_i) \mapsto (\exists i : q_i) \\ (p \mapsto q) \wedge (r \mathbf{U} s) &\Rightarrow ((p \wedge r) \mapsto (q \wedge r) \vee s) \end{aligned}$$

the last rule known in UNITY lingo as “progress-safety-progress” (PSP).

The main reason for introducing \mathbf{E} (instead of just working with \mapsto and \mathbf{U}) is that, unlike \mapsto , \mathbf{E} properties can be composed, using the union rule:

$$(p \mathbf{E} q \text{ in } A) \wedge (p \mathbf{U} q \text{ in } B) \Rightarrow (p \mathbf{E} q \text{ in } A|B)$$

The main purpose of this paper is to define a replacement for \mapsto that has a similar union rule (under suitable semantic constraints on A and B), while preserving the composition rules of (1).

It is not hard to see that, for the purpose of showing progress properties of programs under union, the \mathbf{U} and \mathbf{E} properties of a program are a fully abstract semantics. Therefore, we define \sim (congruence) and \triangleleft (containment) of programs in the obvious way (\mathbf{op} : ranges over $\{\mathbf{U}, \mathbf{E}\}$):

$$\begin{aligned} A \sim B &\equiv (\forall p, q, \mathbf{op} : (p \mathbf{op} q \text{ in } A) \Leftrightarrow (p \mathbf{op} q \text{ in } B)) \\ A \triangleleft B &\equiv (A|B \sim B) \end{aligned}$$

An example of the advantage of using semantic notions (as opposed to equality and subset) can be seen when we extend guarding to programs

$$(A \text{ if } p) \equiv \{(a \text{ if } p) \text{ s.t. } a \in A\}$$

; we then have $(A \text{ if } p) \triangleleft A$.

We will make frequent use of the following two properties derived from **U**:

$$\begin{aligned} (p \text{ stable in } A) &\equiv (p \text{ U false in } A) \\ \langle A \rangle.p &\equiv (\forall q : [p \Rightarrow q] \wedge (q \text{ stable in } A) : q) \end{aligned}$$

$(p \text{ stable in } A)$ holds if no transition of A can falsify p . $\langle A \rangle.p$ is the strongest predicate that both contains p and is stable in A ; i.e. it describes the set of all states reachable from p -states via a (possibly empty) sequence of A transitions.

2.1 Continuity

Some of our results make use of the following semantic property of finite programs. Let ch range over totally ordered sets of predicates, and define

$$(A \text{ cont}) \equiv (\forall ch : (\forall p \in Ch : (p \text{ E } q \text{ in } A)) \Rightarrow ((\exists p \in Ch : p) \text{ E } q \text{ in } A))$$

Intuitively, $(A \text{ cont})$ means that whenever A guarantees atomic progress to (i.e., ensurity of) a goal from each of a weakening sequence of starting points, it can achieve atomic progress from their disjunction.

Although not all programs are continuous, most programs of interest can be shown to be continuous using the following theorems:

$$(\{f\} \text{ cont}) \tag{2}$$

$$(A \text{ cont}) \wedge (B \text{ cont}) \Rightarrow (A|B \text{ cont}) \tag{3}$$

$$(\forall i, j : (A_i \text{ cont}) \wedge [p_i \wedge p_j \Rightarrow i = j]) \Rightarrow (([i : (A_i \text{ if } p_i)] \text{ cont}) \tag{4}$$

These rules say that any singleton program is continuous, and that continuity is preserved by finite union or arbitrary disjoint union.

3 Achievement

We would like to define a replacement for $\mapsto, \rightsquigarrow$ (“achieves”), such that an achievement property of the consumer is an achievement property of the whole producer-consumer system. The reason that this doesn’t work with \mapsto is that the producer might send another message before the consumer gets a chance to execute. For example, the consumer might be guaranteed to eventually make the channel empty when running in isolation, but not when run in parallel with the producer.

An obvious way to overcome this problem is to define $(p \rightsquigarrow q \text{ in } A)$ so that it holds if, from a p state, A is guaranteed to reach some state reachable from

a q state (i.e., an $\langle A \rangle.q$ state). To make sure that \rightsquigarrow is transitive, we similarly weaken the antecedent p , yielding the proposed definition

$$? (p \rightsquigarrow q \text{ in } A) \equiv (\langle A \rangle.p \mapsto \langle A \rangle.q \text{ in } A)$$

However, this definition is too lenient – because it allows progress “backward in time”, it is incompatible with the PSP theorem. For example, if A is the program $\{x := \text{true}\}$, we would have $(x \rightsquigarrow \neg x \text{ in } A)$ and $(x \mathbf{U} \text{false in } A)$; the PSP theorem then yields $(x \rightsquigarrow \text{false in } A)$, which is not what we want.

The remedy is to build into the definition of \rightsquigarrow a quantification over all possible \mathbf{U} properties with which it might be combined (using PSP). This leads to the definition

$$(p \rightsquigarrow q \text{ in } A) \equiv (\forall r, s : (r \mathbf{U} s \text{ in } A) \Rightarrow (\langle A \rangle.(s \vee (r \wedge p)) \mapsto \langle A \rangle.(s \vee (r \wedge q)) \text{ in } A))$$

(the antecedent has again been weakened to recover transitivity.) To a good approximation, $(p \rightsquigarrow q \text{ in } A)$ means that, for any p -state s_0 ,

$$\langle A \rangle.s_0 \mapsto \langle A \rangle.(q \wedge \langle A \rangle.s_0)$$

i.e., from any state reachable from s_0 (via A), A is guaranteed to reach a state that is reachable from s_0 via a path that contains a q -state.

The definition of \rightsquigarrow is obviously much too complex to use directly. Thankfully, we don’t need to, because we can reason about \rightsquigarrow pretty much as we reason about \mapsto . In particular, it satisfies the analogues of (1):

$$(p \mapsto q) \Rightarrow (p \rightsquigarrow q) \tag{5}$$

$$(p \rightsquigarrow q) \wedge (q \rightsquigarrow r) \Rightarrow (p \rightsquigarrow r) \tag{6}$$

$$(\forall i : p_i \rightsquigarrow q_i) \Rightarrow ((\exists i : p_i) \rightsquigarrow (\exists i : q_i)) \tag{7}$$

$$(p \rightsquigarrow q) \wedge (r \mathbf{U} s) \Rightarrow ((p \wedge r) \rightsquigarrow (q \wedge r) \vee s) \tag{8}$$

We also need a way to get from \rightsquigarrow back to \mapsto :

$$(p \rightsquigarrow q) \wedge (q \text{ stable}) \Rightarrow (p \mapsto q) \tag{9}$$

That is, any achievement property whose target is stable is also a leads-to property. We argue that these are the only leads-to properties that really matter, since progress to a predicate that is not stable might never be witnessed by an asynchronous observer. If one accepts this argument, then \rightsquigarrow would appear to be at least as good as \mapsto .

The main advantage of achievement is that it has the following powerful composition property: define $(A \text{ dec } B)$ (“ A is decoupled from B ”) and $(A \trianglelefteq B)$ (“ A is a decoupled component of B ”) as follows (where \mathbf{op} ranges over the operators \mathbf{E}, \mathbf{U}):

$$(A \text{ dec } B) \equiv (\forall p, q, \mathbf{op} : (p \mathbf{op} q \text{ in } A) \Rightarrow (\langle B \rangle.p \mathbf{op} \langle B \rangle.q \text{ in } A))$$

$$(A \trianglelefteq B) \equiv (A \triangleleft B) \wedge (A \text{ dec } B)$$

; then we have

$$(p \rightsquigarrow q \text{ in } A) \wedge (A \trianglelefteq B) \Rightarrow (p \rightsquigarrow q \text{ in } B) \quad (10)$$

In other words, if A is a decoupled component of B , then every achievement property of A is also an achievement property of B . Put differently, working with achievement, we can choose which decoupled component is the next one to execute. Clearly, this does not hold for progress.

4 Decoupling

Like \rightsquigarrow , the definition of **dec** is too complicated to use directly. Fortunately, decoupling can be established using the following (incomplete) set of rules:

$$(A|B \text{ dec } A) \quad (11)$$

$$(\forall i : (A_i \text{ dec } B)) \Rightarrow ((|i : A_i) \text{ dec } B) \quad (12)$$

$$(\forall i : (A \text{ dec } B_i)) \wedge (A \text{ cont}) \Rightarrow (A \text{ dec } (|i : B_i)) \quad (13)$$

$$(A \text{ dec } B) \wedge (p, \neg p \text{ stable in } B) \Rightarrow ((A \text{ if } p) \text{ dec } B) \quad (14)$$

$$(\forall i : (A \text{ dec } (B \text{ if } p_i))) \wedge [(\exists i : p_i)] \Rightarrow (A \text{ dec } B) \quad (15)$$

$$(\neg p \text{ stable in } B) \Rightarrow ((A \text{ if } p) \text{ dec } (B \text{ if } \neg p)) \quad (16)$$

It turns out to be useful to consider explicitly the property $(A \text{ dec } A|B)$, which we abbreviate $(A \text{ wdec } B)$ (“ A is weakly decoupled from B ”). (Note that $A \triangleleft B \wedge (A \text{ wdec } B) \Rightarrow (A \trianglelefteq B)$.) As a rule of thumb, two programs whose interactions are free of race conditions are weakly decoupled from each other, while $(A \text{ dec } B)$ means that, in addition, B cannot send information directly to A . Some useful rules for establishing weak decoupling are the following:

$$(A \text{ dec } B) \Rightarrow (A \text{ wdec } B) \quad (17)$$

$$(\forall i, j : (A_i \text{ wdec } B) \wedge (A_i \text{ wdec } A_j)) \Rightarrow ((|i : A_i) \text{ wdec } B) \quad (18)$$

$$(\forall i : (A \text{ wdec } B_i)) \wedge (A \text{ cont}) \Rightarrow (A \text{ wdec } (|i : B_i)) \quad (19)$$

$$(A \text{ wdec } B) \wedge (p \text{ stable in } B) \Rightarrow ((A \text{ if } p) \text{ wdec } B) \quad (20)$$

$$(\forall i : (A \text{ wdec } (B \text{ if } p_i))) \wedge [(\exists i : p_i)] \Rightarrow (A \text{ wdec } B) \quad (21)$$

$$((A \text{ if } p) \text{ wdec } (B \text{ if } \neg p)) \quad (22)$$

As these rules show, decoupling has a better left union rule ((12) vs. (18)), which is why we work with decoupling whenever possible. For example, in a producer-consumer system, producers are decoupled from consumers, while consumers are only weakly decoupled from producers. This means that we can allow race conditions in the producer, while keeping it a decoupled component of the system, but not in the consumer (except under unusual circumstances).

For singleton programs, decoupling can be established with the following theorems:

$$(\exists p, q : [p \vee q] \wedge [p; g; f \Rightarrow f; g] \wedge [q; g \Rightarrow 1]) \Rightarrow (\{f\} \text{ dec } \{g\}) \quad (23)$$

$$\begin{aligned}
(\exists p, q, r : [p \vee q \vee r] \wedge [r \Rightarrow f] \wedge [p; g; f \Rightarrow f; g] \wedge [q; g \Rightarrow 1]) & \quad (24) \\
\Rightarrow (\{f\} \mathbf{wdec} \{g\}) &
\end{aligned}$$

The hypothesis of (23) says that g right-commutes with f from every state from which g can possibly change the state; the hypothesis of (24) says that g right-commutes with f from every state from which g can possibly change the state and f necessarily changes the state. For example, if f and g interact only through a FIFO channel on which f sends and g receives (both asynchronously), then $(\{f\} \mathbf{dec} \{g\})$ and $(\{g\} \mathbf{wdec} \{f\})$. Related forms of commutativity are studied in [13].

5 Example — Loosely-coupled programs

A *loosely-coupled* program [10] is one in which (1) every transition is total and deterministic, and (2) from any state from which two transitions can change the state, the transitions commute. (Dataflow networks [7] are the most familiar example.) In such a program, every transition is weakly decoupled from every other (by (24)); since singletons are continuous (by (2)), each transition is weakly decoupled from the rest of the system (by (19)). Grouping transitions arbitrarily into processes, each process is a decoupled component (by (18)). Thus, in reasoning about a system, we can choose, at each state, any enabled process to be the next one to execute. Proofs based on this are usually simpler than using the fixed point characterization of [7], since we can often reason about simple (first-order) predicates, instead of having to deal with message sequences.

As a concrete example of this kind of reasoning, consider the following loosely-coupled version of the producer-consumer system described in the introduction. Let ch be a FIFO channel, n a natural counter, let $ch!m$ (resp. $ch?m$) be the actions that send (resp. receive) the message m along the channel ch , and define

$$\begin{aligned}
P &= \{(n := n - 1; (\exists f : x := f.x; ch!f) \mathbf{if} \ n > 0)\} \\
C &= \{((\exists f : ch?f; y := f.y) \mathbf{if} \ ch \neq \langle \rangle)\} \\
clean &\equiv x = y \wedge ch = \langle \rangle \\
mid &\equiv (\exists f : x = f.y \wedge ch = \langle f \rangle)
\end{aligned}$$

We can prove ($clean \mapsto clean \wedge n = 0$ **in** $P|C$) as follows:

- 1) $clean \wedge n = N > 0 \mapsto mid \wedge n < N$ **in** P def P
- 2) $clean \wedge n = N > 0 \rightsquigarrow mid \wedge n < N$ **in** P 1, (5)
- 3) $(P \sqsubseteq P|C)$ (24)
- 4) $clean \wedge n = N > 0 \rightsquigarrow mid \wedge n < N$ **in** $P|C$ 2, 3, (10)
-)
- 5) $mid \wedge n < N \mapsto clean \wedge n < N$ **in** C def C
- 6) $mid \wedge n < N \rightsquigarrow clean \wedge n < N$ **in** C 5, (5)
- 7) $(C \sqsubseteq P|C)$ (24)
- 8) $mid \wedge n < N \rightsquigarrow clean \wedge n < N$ **in** $P|C$ 6, 7, (10)
-)
- 9) $clean \wedge n = N > 0 \rightsquigarrow clean \wedge n < N$ **in** $P|C$ 4, 8, (6)
- 10) $clean \wedge n = N \rightsquigarrow clean \wedge n = 0$ **in** $P|C$ 9, (6), induction
- 11) $clean \rightsquigarrow clean \wedge n = 0$ **in** $P|C$ 10, (7)
- 12) $((clean \wedge n = 0)$ **stable in** $P|C)$ def P, C
- 13) $clean \mapsto clean \wedge n = 0$ **in** $P|C$ 11, 12, (9)

6 Asynchronous Safety

Invariants (or, more generally, stable predicates) play a key role in program development. However, asynchrony can make invariants unreasonably complicated. Instead of working with real invariants, we can work with predicates that are reestablished by decoupled components. Because decoupled components can be assumed to execute immediately, these predicates are almost as good as real invariants. The component that reestablishes the invariant is called a “sweeper”, because it cleans up after other components.

Sweepers are defined as follows:

$$(A \text{ sw } B \text{ to } p) \equiv (A \sqsubseteq B) \wedge (\langle B \rangle.p \rightsquigarrow p \text{ in } A)$$

Note that sweeping generalizes stability, i.e.,

$$(p \text{ stable in } A) \Leftrightarrow (1 \text{ sw } A \text{ to } p)$$

A key property of stability is that a predicate is stable in a union of components if it is stable in each component. Sweeping enjoys similar compositionality:

$$(A \text{ cont}) \wedge (\forall i : (A \text{ sw } B_i \text{ to } p)) \Rightarrow ((|i : A) \text{ sw } (|i : B_i) \text{ to } p) \quad (25)$$

The other key property of stability is that it can be combined with progress (or achievement) using a special case of the PSP rule. The corresponding property for sweepers is

$$(A \text{ sw } B \text{ to } p) \wedge (q \rightsquigarrow r \text{ in } B) \Rightarrow (p \wedge q \rightsquigarrow p \wedge \langle A \rangle.r \text{ in } B) \quad (26)$$

In most situations, workers can run far ahead of sweepers, and we don’t want to have to prove $(\langle B \rangle.p \rightsquigarrow p \text{ in } A)$ directly, because $\langle B \rangle.p$ may be

complicated; we would rather sweep up after a single transition of B . In general, if $(A \text{ wdec } B)$ is established using the the rules of section 4, then

$$(p \text{ U } q \text{ in } B) \wedge (A \leq B) \wedge (q \rightsquigarrow p \text{ in } A) \Rightarrow (A \text{ sw } B \text{ to } p)$$

6.1 Example

We modify the producer-consumer example slightly so that termination is caused by a separate component (instead of using a counter in the producer):

$$\begin{aligned} P0 &= \{(\exists f : x := f.x; ch!f) \text{ if } \neg stop\} \\ P1 &= \{stop := true\} \\ P &= P0|P1 \\ C &= \{(\exists f : ch?f; y := f.y) \text{ if } ch \neq \langle \rangle\} \\ clean &\equiv (x = y \wedge ch = \langle \rangle) \\ mid &\equiv (\exists f : x = f.y \wedge ch = \langle f \rangle) \end{aligned}$$

The proof from section 5 does not work here, because there is no state variable n to record progress. However, we can instead use a sweeper proof:

$$\begin{array}{llll} 1) & (C \text{ wdec } P0, P1) & & (24) \\ 2) & clean \quad \text{U} \quad mid & \text{in } P0 & \text{def } P0 \\ 3) & mid \quad \mapsto \quad clean & \text{in } C & \text{def } C \\ 4) & mid \quad \rightsquigarrow \quad clean & \text{in } C & (5) \\ 5) & C \text{ sw } P0 \text{ to } clean & & 1, 2, 4 \\ &) & & \\ 6) & clean \quad \text{U} \quad false & \text{in } P1 & \text{def } P1 \\ 7) & false \quad \rightsquigarrow \quad clean & \text{in } C & (5) \\ 8) & C \text{ sw } P1 \text{ to } clean & & 1, 6, 7 \\ &) & & \\ 9) & C \text{ sw } P \text{ to } clean & & 5, 8, (25) \\ 10) & true \quad \mapsto \quad stop & \text{in } P|C & \text{def } P0 \\ 11) & true \quad \rightsquigarrow \quad stop & \text{in } P|C & 10, (5) \\ 12) & clean \quad \rightsquigarrow \quad clean \wedge \langle C \rangle.stop & \text{in } P|C & 9, 11, (26) \\ 13) & (stop \text{ stable in } C) & & \text{def } C \\ 14) & clean \quad \rightsquigarrow \quad clean \wedge stop & \text{in } P|C & 12, 13 \\ 15) & (clean \wedge stop \text{ stable in } P|C) & & \text{def } P, C \\ 16) & clean \quad \mapsto \quad clean \wedge stop & \text{in } P|C & 14, 15, (9) \end{array}$$

7 Caveats

While achievement has few disadvantages wrt. leads-to in the context of UNITY-like program development, it does have one disadvantage worth mentioning: unlike linear-time properties, achievement is not preserved by program refine-

ment¹, so to use a refinement step, it is first necessary to convert achievement properties back to leads-to. This is hardly surprising; related properties like serializability suffer from the same problem.

A minor annoyance in the theory is the continuity requirement. The definition of $(A \text{ dec } B)$ is of the form “if A has this property, it also has that property”; when the property is an **E** property, there is no way in the logic to make sure that the “that” property is being guaranteed by the same transition that guarantees the “this” property (even though it is in most practical cases). This is a minor price to pay for a theory that works entirely at the level of properties, instead of transitions.

An more serious limitation of the theory is shown in the following example. Suppose we have two producer-consumer systems, A producing for B , and C producing for D . Suppose also that these systems multiplex their communications on a shared channel. Sweeper compositionality lets us prove

$$(B \text{ sw } A|B \text{ to } p) \wedge (D \text{ sw } C|D \text{ to } p) \Rightarrow (B|D \text{ sw } A|B|C|D \text{ to } p)$$

(assuming we’ve correctly labelled multiplexed messages so that B and D don’t try to receive the same messages) so things are fine from the sweeper standpoint. However, we would like to prove something stronger, namely $(A|B|D \text{ dec } C|D)$, which would, in effect, allow us to pretend that the communication is not multiplexed. (This problem came up in trying (with Rajeev Joshi) to use sweepers to prove the correspondence of loose and tight executions in Seuss[11].) We do not know how to strengthen the theory to handle this situation.

Finally, because it is fundamentally about progress, the theory is highly asymmetric with respect to time. Decoupled components work as weak “left-movers”; we can always pretend they happen earlier. They can be composed precisely because all of them are moving in the same direction. Reduction theorems such as [4], on the other hand, allow both left- and right-movers, so message transmissions can be moved later (instead of just moving receptions earlier).

8 Conclusions

We have argued that achievement has some desirable properties that make it technically superior to leads-to, particularly when reasoning about asynchronous programs. More generally, we have shown that it is possible to weaken linear-time operators to branching-time operators operators so as to make them more robust to asynchrony, without changing the essential structure of the logic.

¹For example, $(\text{true} \rightsquigarrow x \text{ in } \{(x := \text{false}), (x := \text{any})\})$, but $\neg(\text{true} \rightsquigarrow x \text{ in } \{(x := \text{false})\})$.

9 Acknowledgements

This work was originally inspired by Jay Misra's paper [10]; it has benefitted from insightful discussions with Jay Misra, J. R. Rao, and Rajeev Joshi, and from the insightful comments of the anonymous referee.

References

- [1] K. M. Chandy and J. Misra. *Parallel Program Design, A Foundation*. Addison-Wesley Co., Inc., Reading, MA, 1988
- [2] E. Cohen. *Modular Progress Proofs of Asynchronous Programs*. PhD. thesis, University of Texas at Austin (1993).
- [3] E. Cohen. *Separation and Reduction*. In Mathematics of Program Construction 2000 (Springer-Verlag).
- [4] E. Cohen, L. Lamport. *Reduction in TLA*. In CONCUR98 (Springer-Verlag).
- [5] E. Dijkstra and C. Scholten. *Predicate transformers and Program Semantics*. (Springer-Verlag).
- [6] K. P. Eswaran, J. N. Gray, R. A. Lorie, I. L. Traiger. *The notions of consistency and predicate locks in a database system*. CACM, 19(11):624-633, 1976.
- [7] G. Kahn. *The semantics of a simple language for parallel programming*. In *Proceedings of IFIP Congress '74* (North-Holland, 1974)
- [8] S. Katz, D. Peled. *Verification of Distributed Programs using Representative Interleaving Sequences*. Distributed Computing (1992) 6:107-120 (Springer-Verlag).
- [9] R. J. Lipton. *Reduction: A Method of Proving Properties of Parallel Programs*. CACM 18(12):717-721, 1975.
- [10] J. Misra. *Loosely Coupled Programs*. In *Parallel Architectures and Languages Europe*, pages 1—26, June 1991.
- [11] J. Misra. *A Discipline of Multiprogramming*.
- [12] J. Pachl. *A simple proof of a completeness result for leads-to in the UNITY logic*. IPL, January 1992.
- [13] J. R. Rao. *Extensions of the UNITY Methodology, Compositionality, Fairness and Probability in Parallelism*. Springer LNCS #908 (1995)
- [14] A. Valmari. *Stubborn Sets for Reduced State Space Generation*. 10th International Conference on Application and Theory of petri Nets, Bonn 1989 (2) pp 1-22.