2012

Fairness for Transactional Events

Edward Amsden
Rochester Institute of Technology

Matthew Fluet
Rochester Institute of Technology

Recommended Citation
Amsden E., Fluet M. (2012) Fairness for Transactional Events. In: Gill A., Hage J. (eds) Implementation and Application of Functional Languages. IFL 2011. Lecture Notes in Computer Science, vol 7257. Springer, Berlin, Heidelberg

Fairness for Transactional Events
Technical Report

Edward Amsden and Matthew Fluet

Computer Science Department
Rochester Institute of Technology, Rochester NY 14623
{eca7215,mtf}@cs.rit.edu

Abstract. *Transactional events* are a recent concurrency abstraction that combines first-class synchronous message-passing events with all-or-nothing transactions. Transactional events provide both a sequencing combinator, which permits the description of behaviors in which multiple potential synchronization actions (including communications between threads) either all occur in sequence or none of them occur, and a non-deterministic choice combinator, which permits the description of behaviors in which exactly one of a set of potential synchronization actions occurs. While prior work gave a semantics and an implementation for transactional events, it provided no guarantees about which of the many non-deterministic executions might be exhibited by a program.

For concurrent systems, like transactional events, it is natural to expect certain *fairness* conditions to hold on executions. Intuitively, fairness guarantees that any system component that could (sufficiently often) make progress does, in fact, make progress. In this work, we investigate fairness for transactional events. We give a rigorous definition of fair program executions in transactional events, describe a refined operational semantics that guarantees fair executions, and discuss restrictions and assumptions necessary for the correctness of an implementation based on the refined semantics.

This is a companion technical report, providing additional commentary and proof details, to a paper [1] appearing in *Implementation and Application of Functional Languages: 23rd International Symposium (IFL'11)*.

1 Introduction

Concurrent programming can be a difficult task. The non-deterministic nature of a concurrent program's execution makes it difficult to reason about all of the possible behaviors of the program. To manage the complexity of writing and understanding concurrent programs, programmers make use of two enabling methodologies: (1) high-level abstractions of concurrent operations and (2) assumed properties of concurrent systems.

High-level concurrency abstractions allow complex thread interactions to be abstractly packaged and exported, which increases modularity and eases reasoning. For example, Software Transactional Memory (STM) [14, 6] provides

first-class, composable operations that allow a programmer to combine shared-memory operations into an action that can itself be treated as an atomic shared-memory operation. Similarly, Concurrent ML (CML) [12] provides first-class, composable operations that allow a programmer to combine synchronous message-passing operations into an action that can itself be treated as a synchronous message-passing operation. Recently, Transactional Events [3, 4] have been proposed as a concurrency abstraction that combines synchronous message-passing operations with all-or-nothing transactions. The key to the expressive power of Transactional Events is a sequencing combinator that allows a programmer to write an action that contains multiple communications; the action blocks until all of the constituent communications can succeed.

Safety and liveness properties assert statements that are true of all possible executions of a concurrent program. Intuitively, safety asserts that something "bad" never happens, while liveness asserts that something "good" eventually happens. Fairness [5, 2, 8] is a particular liveness property that is important in concurrent programming, although it is often treated informally or assumed implicitly. For example, a concurrent programmer typically assumes a "fair" thread scheduler: all threads in the program will execute, not just some threads. As another example, a concurrent programmer typically assumes that if one thread is attempting to send a message and another thread is attempting to receive the message, then the message will eventually be communicated by the system. In generally, fairness is the property that asserts that any action that is enabled often enough is eventually taken.

There are many situations in which fairness is a useful guarantee. Imagine a system that is structured using looping threads to handle external input and output, as is often the case with console, GUI, or network applications. Threads that need to process input might nondeterministically choose between several input sources; fairness could guarantee that no input source is ignored forever. As another example, consider a "server" design pattern where one thread controls a shared resource and follows a request-response communication pattern to provide access to the resource; a fair implementation would guarantee that no thread that makes a request is permanently excluded from accessing the resource.

This paper examines fairness for transactional events. Transactional events takes synchronous message-passing on channels as the primitive concurrent operation and provides combinators for sequencing (e.g., "perform one communication *and then* perform another communication") and choosing (e.g., "perform *either* one communication *or* another communication") concurrent operations. Since synchronous message-passing requires matching a sender and a receiver and sequencing and choosing requires examining multiple communications, the enabledness of a transactional event is non-trivial, which gives rise to interesting behaviors with respect to fairness. As a simple example, consider one thread that repeatedly sends on a channel and two threads that repeatedly receive on the channel. It is intuitively unfair to repeatedly match the sending thread with exactly one of the receiving threads, never communicating with the other receiv-

ing thread. We are interested in formalizing this intuition and guaranteeing this behavior in an implementation. We make the following contributions:

- We give an intuitive, yet formal, definition of fairness for transactional events (Section 4) in terms of a high-level operational semantics (Section 3).
- We describe a lower-level operational semantics (Section 5) that refines the high-level operational semantics and demonstrate that executions in the lower-level semantics simulate fair executions (and only fair executions) of the higher-level semantics (Theorems 1 and 2). We do not demonstrate an equivalence of the lower-level semantics and the high-level semantics, since we necessarily exclude some fair high-level executions. However, we argue informally that, since every initial program has a trace in the lower-level semantics, and every lower-level execution simulates a high-level execution, then every initial program will actually produce a high-level execution under an implementation corresponding to the lower-level semantics. Thus, a programmer who reasons about all fair executions in the high-level semantics can be confident that that reasoning applies to any actual execution that is realized by an implementation.
- We discuss an implementation of the lower-level semantics and suggest a property of synchronizing events, which, if statically verified by a programmer, enables the implementation to enforce fairness (Section 6).

2 Background

2.1 Transactional Events

In this section, we introduce transactional events as a Haskell library in the context of Concurent Haskell [10, 9].[1] The basic interface for transactional events is given in Figure 1. The type `Evt a` is the type of an abstract synchronous operation that yields a result of type `a` when synchronized upon. A transactional event only *describes* a synchronous operation (which might include synchronous communication), it does not immediately *perform* the synchronous operation. In order to perform the synchronous operation described by a transactional event `evt`, some thread must perform the I/O action `sync evt`; that is, the `sync` operation takes a transactional event to an I/O action that synchronizes upon the event. Note that synchronization depends upon the state of concurrently synchronizing threads, which is why `sync` has the type `Evt a -> IO a` and not `Evt a -> a`.

The type `SChan a` is the type of a synchronous channel carrying messages of type `a`. The basic events `sendEvt ch m` and `recvEvt ch` correspond to the synchronous operations that send the message `m` over the channel `ch` and that receive a message over the channel `ch`, respectively. Message passing is synchronous, so

[1] We assume that the reader is familiar with Haskell, monadic I/O [11, 9], and Concurrent Haskell [10, 9]; in particular, we will use monadic `do`-notation and the following operations: `putChar :: Char -> IO ()` and `forkIO :: IO () -> IO ThreadId`.

```
data Evt a                                          -- The Evt type

sync      :: Evt a -> IO a                  -- Synchronization

data SChan a                              -- Synchronous channels
newSChan  :: Evt (SChan a)
sendEvt   :: SChan a -> a -> Evt ()
recvEvt   :: SChan a -> Evt a

alwaysEvt :: a -> Evt a                      -- Event combinators
thenEvt   :: Evt a -> (a -> Evt b) -> Evt b
neverEvt  :: Evt a
chooseEvt :: Evt a -> Evt a -> Evt a
```

Fig. 1. The `TxEvent` interface

every send over a channel must be matched by a receive over the same channel; synchronizing on an unmatched communication blocks until there is a matching communication being synchronized upon. The basic event `newSChan` creates new channels; `newSChan` has an event type to enable the useful idiom of creating reply channels that are local to a synchronization. Here is a simple program that creates a channel, forks two threads to send different values on the channel, receives on the channel, and finally prints the character:

```
main = do { ch <- sync newSChan
          ; forkIO (sync (sendEvt ch 'a'))
          ; forkIO (sync (sendEvt ch 'b'))
          ; c <- sync (recvEvt ch)
          ; putChar c }
```

This program should print either `'a'` or `'b'`, since the main thread can synchronize with exactly one of the forked threads. The other forked thread remains blocked, unable to synchronize without a matching receiver. This example highlights the fact that transactional events have an implicit choice, whereby the the matching of senders and receivers is non-deterministic.

Transactional events may be composed in a variety of ways. The `chooseEvt` combinator allows events to be composed as non-deterministic alternatives. The event `evt1 `chooseEvt` evt2` synchronizes as either `evt1` or `evt2`, but must choose exactly one sub-event that can successfully synchronize. For example, the following program should print `'a'`, since the main thread must not choose to synchronize upon the event `recvEvt ch2` which has no matching sender.

```
main = do { ch1 <- sync newSChan
          ; ch2 <- sync newSChan
          ; forkIO (sync (sendEvt ch1 'a'))
          ; c <- sync ((recvEvt ch1) `chooseEvt`
                       (recvEvt ch2))
          ; putChar c }
```

The `thenEvt` combinator allows events to be composed in sequence. The event `evt `thenEvt` f` synchronizes as `evt`, yielding the result `r`, and then synchronizes on the event `f r`, but must successfully synchronize on both sub-events. Note that the second event may depend upon the result of the first event. For example, the following program should print `'a'` and `'b'` (in some non-deterministic order), since the first forked thread is able to send to the second forked thread and the main thread as part of a single synchronization:

```
main = do { ch <- sync newSChan
          ; forkIO (sync ((sendEvt ch 'a') `thenEvt`
                          (\ _ -> sendEvt ch 'b')))
          ; forkIO (do { c <- sync (recvEvt ch)
                       ; putChar c })
          ; c <- sync (recvEvt ch)
          ; putChar c }
```

On the other hand, this program deadlocks, since the forked thread is unable to send to two matching receivers as part of a single synchronization.

```
main = do { ch <- sync newSChan
          ; forkIO (sync ((sendEvt ch 'a') `thenEvt`
                          (\ _ -> sendEvt ch 'b')))
          ; c <- sync (recvEvt ch)
          ; putChar c
          ; c <- sync (recvEvt ch)
          ; putChar c }
```

The event `alwaysEvt e` is an event that may always be successfully synchronized upon to yield `e`, while the event `neverEvt` is an event that may never be successfully synchronized upon. These events may be composed with the event combinators to give rise to sophisticated behaviors. For example, the following derived event implements guarded (or conditional) receive:

```
grecvEvt :: (a -> Bool) -> SChan a -> Evt a
grecvEvt g ch =
  (recvEvt ch) `thenEvt`
  (\ x -> if g x then alwaysEvt x else neverEvt)
```

The event `grecvEvt g ch` corresponds to the synchronous operation that receives only messages that satisfy the guard `g` over the channel `ch`. The `thenEvt` sequences the `recvEvt ch` with either `alwaysEvt x` or `neverEvt`, depending upon the result of `g x`. Since `thenEvt` requires both sub-events to successfully synchronize and `neverEvt` never successfully synchronizes, the whole event may only successfully synchronize when the message received satsifies the guard.

2.2 Fairness

The informal description of transactional events given above begins to suggest some of the possible behaviors of programs. However, among all of the possible

behaviors of a program, there are some which are so counterintuitive and/or undesirable that it is useful to classify them as "illegal" and require an implemenation to not exhibit such behaviors. For example, consider the following program that has three threads, each of which repeatedly prints a different character:[2]

```
main = do { forkIO (forever (putChar 'a'))
          ; forkIO (forever (putChar 'b'))
          ; forever (putChar 'C') }
```

This program has many possible behaviors:

- `aabaaabcacbccca...`
- `acbbababcbbbacb...`
- `cacaaaccccabbca...`
- `aaaaaaaaaaaaaaa...`
- `cbcccbcbcbcbccb...`

Although the first three appear reasonable, the last two might be considered unreasonable, as they seem to suggest that one or more threads are starved (excluded from execution). Nonetheless, many formalisms of concurrency would allow all of the above behaviors.

In order to exclude such behaviors, we can appeal to a notion of *fairness* [5, 2, 8]. Intuitively, the last two behaviors are "unfair", because the starved thread is always able to execute, yet is excluded from execution. In its most general form, fairness asserts that any component of a concurrent system that is enabled often enough is not excluded from execution. More specific notions of fairness arise from different interpretations of "component", "enabled", and "often enough". For the program above, and for concurrent threads without synchronization primitives, fairness essentially describes the behavior of a "fair" thread scheduler that guarantees that no thread is starved: a "component" is a thread and threads are always "enabled" (due to the absence of synchronization primitives) and, hence, are always enabled "often enough".

For transactional events, the notion of fairness is more subtle. On the one hand, it is clear that threads are not always "enabled": a thread performing a synchronization on a `sendEvt` in a program where there is (and will never be) another thread performing a synchronization on a matching `recvEvt` cannot execute; fairness should not demand that this thread eventually executes. On the other hand, in the following program, both of the forked threads have many opportunities to synchronize with the main thread:

```
main = do { ch <- sync newSchan
          ; forkIO (forever (sync (sendEvt ch 'a')))
          ; forkIO (forever (sync (sendEvt ch 'b')))
          ; forever (do { c <- sync (recvEvt ch)
                        ; putChar c }) }
```

[2] The function `forever :: Monad m => m a -> m b`, repeats a monadic action infinitely.

Therefore, a notion of fairness for transactional events should demand that both of the forked threads eventually execute and that the program print some interleaving of `'a'`s and `'b'`s (and that the program does not print exclusively `'a'`s or exclusively `'b'`s).

3 Semantics

In this section, we review the original high-level operational semantics for transactional events.

Syntax Expressions naturally fall into one of four categories: standard pure functional language expressions (variables, abstractions, applications, ...), special constants (characters c, thread identifiers θ, and channel names κ), `Evt` combinators, and `IO` combinators:

Expressions

$e ::= x \mid \lambda x.e_b \mid e_f\ e' \mid \ldots$	pure functional language expressions
$\mid\ c \mid \theta \mid \kappa$	special constants
$\mid$ `alwaysEvt` $e' \mid$ `thenEvt` $e_{evt}\ e_f$	`Evt` combinators
$\mid$ `neverEvt` $\mid$ `chooseEvt` $e_{evtl}\ e_{evtr}$	
$\mid$ `newSChan` $\mid$ `recvEvt` $e_k \mid$ `sendEvt` $e_k\ e'$	
$\mid$ `unitIO` $e' \mid$ `bindIO` $e_{io}\ e_f$	`IO` combinators
$\mid$ `getChar` $\mid$ `putChar` $e_c \mid$ `forkIO` $e_{io} \mid$ `sync` e_{evt}	

Operational Semantics The essence of the operational semantics is to interpret sequential expressions, `Evt` expressions, and `IO` expressions as separate sorts of computations. This is expressed by three levels of evaluation: sequential evaluation of pure expressions, synchronous evaluation of transactional events, and concurrent evaluation of IO threads. The bridge between the `Evt` and `IO` computations is synchronization, which moves threads from concurrent evaluation to synchronous evaluation and back to concurrent evaluation.

Sequential Evaluation ($e \hookrightarrow e'$) The "lowest" level of evaluation is the sequential evaluation of pure functional language expressions. Unsurprisingly, the sequential evaluation relation is entirely standard and thus omitted, although it is expected that sequential evaluation can express recursion and recursively defined `Evt` and `IO` computations. We note that the order of evaluation for pure expressions (whether call-by-value, call-by-name, or call-by-need) has no real impact on the behavior of transactional events or the definition of fairness.

Synchronous Evaluation ($\mathcal{E} \rightsquigarrow \mathcal{E}'$) The "middle" level of evaluation is synchronous evaluation of transactional events (see Fig. 2). A *synchronization group* is a set of *synchronizing events*, which are themselves pairs of thread identifiers and `Evt` expressions. Intuitively, the relation $\{\langle\theta_1, e_{evt1}\rangle, \ldots\} \rightsquigarrow \{\langle\theta_1, e'_{evt1}\rangle, \ldots\}$

$$\begin{array}{l|l} \textit{Synchronizing Event} \quad E ::= \langle \theta, e_{evt} \rangle & \textit{Synchronous Evaluation Contexts} \\ \textit{Synchronization Group} \quad \mathcal{E} ::= \{E, \ldots\} & M^{Evt} ::= [] \mid \texttt{thenEvt}\ M^{Evt}\ e_f \end{array}$$

$$\textsc{EvtEval}\ \frac{e \hookrightarrow e'}{\mathcal{E} \uplus \{\langle \theta, M^{Evt}[e] \rangle\} \rightsquigarrow \mathcal{E} \uplus \{\langle \theta, M^{Evt}[e'] \rangle\}} \qquad \textsc{EvtThenAlways}\ \frac{}{\mathcal{E} \uplus \{\langle \theta, M^{Evt}[\texttt{thenEvt}\ (\texttt{alwaysEvt}\ e')\ e_f] \rangle\} \rightsquigarrow \mathcal{E} \uplus \{\langle \theta, M^{Evt}[e_f\ e'] \rangle\}}$$

$$\textsc{EvtChooseLeft}\ \frac{}{\mathcal{E} \uplus \{\langle \theta, M^{Evt}[\texttt{chooseEvt}\ e_{evtl}\ e_{evtr}] \rangle\} \rightsquigarrow \mathcal{E} \uplus \{\langle \theta, M^{Evt}[e_{evtl}] \rangle\}} \qquad \textsc{EvtChooseRight}\ \frac{}{\mathcal{E} \uplus \{\langle \theta, M^{Evt}[\texttt{chooseEvt}\ e_{evtl}\ e_{evtr}] \rangle\} \rightsquigarrow \mathcal{E} \uplus \{\langle \theta, M^{Evt}[e_{evtr}] \rangle\}}$$

$$\textsc{EvtNewSChan}\ \frac{\kappa' \text{ fresh}}{\mathcal{E} \uplus \{\langle \theta, M^{Evt}[\texttt{newSChan}] \rangle\} \rightsquigarrow \mathcal{E} \uplus \{\langle \theta, M^{Evt}[\texttt{alwaysEvt}\ \kappa'] \rangle\}}$$

$$\textsc{EvtSendRecv}\ \frac{}{\mathcal{E} \uplus \{\langle \theta_s, M_s^{Evt}[\texttt{sendEvt}\ \kappa\ e'] \rangle, \langle \theta_r, M_r^{Evt}[\texttt{recvEvt}\ \kappa] \rangle\} \rightsquigarrow \mathcal{E} \uplus \{\langle \theta_s, M_s^{Evt}[\texttt{alwaysEvt}\ ()] \rangle, \langle \theta_r, M_r^{Evt}[\texttt{alwaysEvt}\ e'] \rangle\}}$$

Fig. 2. Dynamic semantics – Synchronous evaluation

means that the events $e_{evt1}, \ldots$ make one step towards synchronization by evaluating to the events $e'_{evt1}, \ldots$. All of the synchronous evaluation rules nondeterministically choose one or more events for a step of evaluation.

The EvtEval rule implements the sequential evaluation of an expression in the active position. The EvtThenAlways rule implements sequential composition in the `Evt` monad. The EvtChooseLeft and EvtChooseRight rules implement a non-deterministic choice between events. The EvtNewSChan rule allocates a new channel name; note that the freshness of κ' is with respect to the entire program state.[3] The EvtSendRecv rule implements the two-way rendezvous of communication via a channel; the transition replaces the `sendEvt` and `recvEvt` events with `alwaysEvt` events.

We define $\mathsf{SYNCABLE}(\mathcal{E})$, a predicate asserting that the non-empty synchronization group $\mathcal{E}$ may successfully synchronize by evaluating to a configuration in which all events are `alwaysEvt`s.

$$\mathsf{SYNCABLE}(\{\langle \theta_1, e_{evt1} \rangle, \ldots\}) \stackrel{\text{def}}{=} \exists e'_1, \ldots.\ \{\langle \theta_1, e_{evt1} \rangle, \ldots\} \rightsquigarrow^* \{\langle \theta_1, \texttt{alwaysEvt}\ e'_1 \rangle, \ldots\}$$

Concurrent Evaluation ($\mathcal{P} \xrightarrow{a} \mathcal{P}'$) The "highest" level of evaluation is concurrent evaluation of threads (see Fig. 3). A *thread soup* is a set of *IO threads*, which are

[3] This freshness condition could be formalized by a synchronous evaluation relation of the form $\mathcal{K}; \mathcal{E} \rightsquigarrow \mathcal{K}'; \mathcal{E}'$, where $\mathcal{K}$ is a set of allocated channel names.

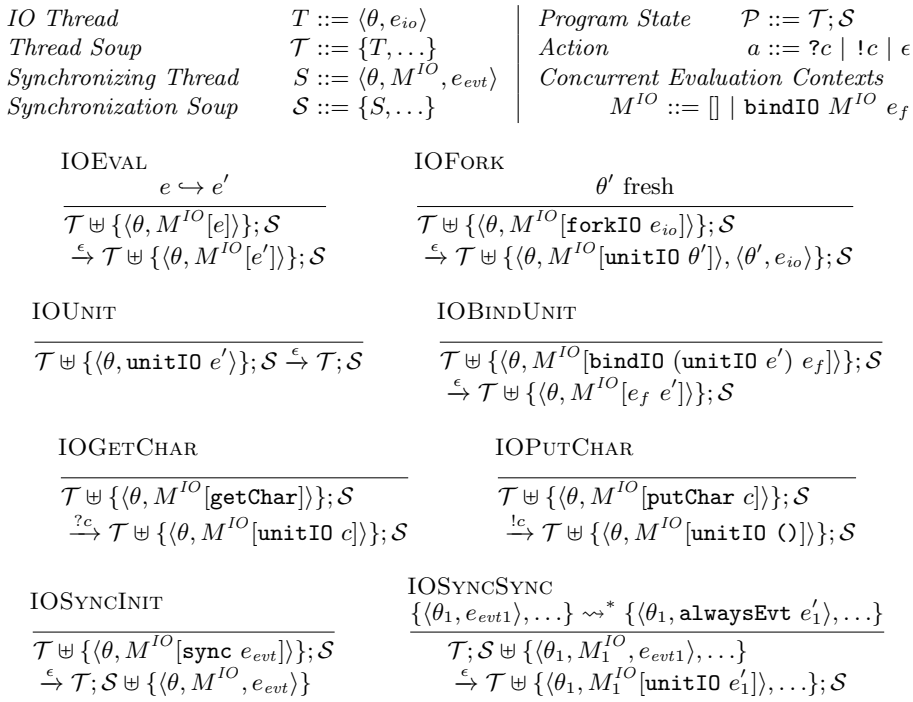

Fig. 3. Dynamic semantics – Concurrent evaluation

themselves pairs of thread identifiers and `IO` expressions. A *synchronization soup* is a set of *synchronizing threads*, which correspond to IO threads that are actively synchronizing and are themselves triples of a thread identifier, an `IO` evaluation context, and an `Evt` expression. Finally, a *program state* pairs a thread soup and a synchronization soup. *Actions* represent the input/output behavior of the program. The observable actions correspond to reading a character c from standard input ($\texttt{?}c$) or writing a character c to standard output ($\texttt{!}c$). The silent action ϵ indicates no observable input/output behavior. In a real language, there would be many other observable I/O actions.ll of the concurrent evaluation rules non-deterministically choose one or more threads for a step of evaluation and are labeled with an action.

The IOEval rule implements the sequential evaluation of an expression in the active position. The IOFork rule implements thread creation by adding a new IO thread to the thread soup; the fresh thread identifier of the child thread is returned to the parent thread.[4] The IOUnit rule implements thread termination when a thread has evaluated to a `unitIO` action. The IOBindUnit

[4] This freshness condition, along with the freshness condition in the EvtNewSChan rule of the synchronous evaluation relation, could be formalized by a concurrent

rule implements sequential composition in the `IO` monad. The IOGetChar and IOPutChar rules perform the appropriate labeled transition, yielding observable actions.

The IOSyncInit and IOSyncSync rules implement event synchronization. The IOSyncInit rule initiates event synchronization by changing an IO thread into a synchronizing thread. The IOSyncSync rule completes event synchronization by selecting some non-empty collection of synchronizing threads, passing the event expressions to the synchronous evaluation relation, which takes *multiple transitions* to a configuration in which all events are `alwaysEvt`s, and resuming all of the synchronizing threads as IO threads with their synchronization results.

Note that the IOSyncInit and IOSyncSync rules have silent actions. Synchronization is not observable, though it may unblock a thread so that subsequent I/O actions are observed. Also note that the IOSyncSync rule takes multiple synchronous evaluation steps in a single concurrent evaluation step; this guarantees that synchronization executes "atomically," although the synchronization of a single event is not "isolated" from the synchronizations of other events. (Indeed, it is imperative that multiple events synchronize simultaneously in order to enable synchronous communication along channels.)

4 Fairness

Intuitively, fairness is a property that asserts that every thread that could make progress does, in fact, make progress. Alternatively, unfairness is a property that asserts that some thread that could make progress does not make progress. Hence, fairness and unfairness are a properties of a program's execution, rather than a property of a program's state. We start by formalizing a representation of a program's execution and then introduce two distinct notions of fairness.

Traces A *program trace* is a representation of a program's execution. Since many interesting concurrent programs do not terminate, a program trace must represent both terminating and non-terminating executions. Furthermore, a program trace must represent a maximal execution, where a terminating execution is witnessed by a terminal program state that cannot evolve. A program trace $\mathcal{P} \Uparrow$ is defined using a coinductive relation, in order to represent both terminating and non-terminating executions.

$$\textsc{Term}\ \frac{\neg\exists a, \mathcal{P}'.\ \mathcal{P} \xrightarrow{a} \mathcal{P}'}{\mathcal{P} \Uparrow}\ \mathsf{co} \qquad \textsc{Step}\ \frac{\mathcal{P} \xrightarrow{a} \mathcal{P}' \qquad \mathcal{P}' \Uparrow}{\mathcal{P} \Uparrow}\ \mathsf{co}$$

The Term rule indicates that $\mathcal{P}$ is a terminal program state, while the Step rule indicates that $\mathcal{P}$ may evolve to a new program state.

evaluation relation of the form $\Theta; \mathcal{K}; \mathcal{T} \xrightarrow{a} \Theta'; \mathcal{K}'; \mathcal{T}'$, where Θ is a set of allocated thread identifiers and $\mathcal{K}$ is a set of allocated channel names.

Since a program trace may be seen to define a finite or infinite sequence of program steps and program states, it is convenient to index a program trace in order to extract the i^{th} program step or program state. These recursive, partial, indexing operations $I^{\rightarrow}(\mathcal{P} \Uparrow, i)$ for steps and $I^{\mathcal{P}}(\mathcal{P} \Uparrow, i)$ for states, are defined as follows:

$$I^{\rightarrow}\left(\frac{\textsc{Step}\ \mathcal{P} \xrightarrow{a} \mathcal{P}' \qquad \mathcal{P}' \Uparrow}{\mathcal{P} \Uparrow}\ \text{co}, 0\right) = \mathcal{P} \xrightarrow{a} \mathcal{P}'$$
$$I^{\rightarrow}\left(\frac{\textsc{Step}\ \mathcal{P} \xrightarrow{a} \mathcal{P}' \qquad \mathcal{P}' \Uparrow}{\mathcal{P} \Uparrow}\ \text{co}, i+1\right) = I^{\rightarrow}(\mathcal{P}' \Uparrow, i)$$

$$I^{\mathcal{P}}\left(\mathcal{P} \Uparrow, 0\right) = \mathcal{P}$$
$$I^{\mathcal{P}}\left(\frac{\textsc{Step}\ \mathcal{P} \xrightarrow{a} \mathcal{P}' \qquad \mathcal{P}' \Uparrow}{\mathcal{P} \Uparrow}\ \text{co}, i+1\right) = I^{\mathcal{P}}(\mathcal{P}' \Uparrow, i)$$

As a convenience, we define $I^{\mathcal{T}}(\mathcal{P} \Uparrow, i)$ and $I^{\mathcal{S}}(\mathcal{P} \Uparrow, i)$ to extract the i^{th} thread soup and the i^{th} synchronization soup, respectively, from a program trace:

$$I^{\mathcal{T}}(\mathcal{P} \Uparrow, i) = \mathcal{T} \qquad \text{where } \mathcal{T}; \mathcal{S} = I^{\mathcal{P}}(\mathcal{P} \Uparrow, i)$$
$$I^{\mathcal{S}}(\mathcal{P} \Uparrow, i) = \mathcal{S} \qquad \text{where } \mathcal{T}; \mathcal{S} = I^{\mathcal{P}}(\mathcal{P} \Uparrow, i)$$

While a program trace is a representation of a program's complete execution, an *action trace* is a representation of a program's observable input/output behavior. An action trace is a finite or infinite sequence of actions, defined using a coinductive interpretation.

$$\textit{Action Trace} \qquad \mathcal{A} ::\overset{\text{co}}{=} \bullet \mid a : \mathcal{A}$$

We define $\mathit{Acts}(\mathcal{P} \Uparrow)$ as a corecursive, total operation that constructs an action trace from a program trace:

$$\mathit{Acts}\left(\frac{\textsc{Term}\ \neg \exists a, \mathcal{P}'.\ \mathcal{P} \xrightarrow{a} \mathcal{P}'}{\mathcal{P} \Uparrow}\ \text{co}\right) = \bullet$$
$$\mathit{Acts}\left(\frac{\textsc{Step}\ \mathcal{P} \xrightarrow{a} \mathcal{P}' \qquad \mathcal{P}' \Uparrow}{\mathcal{P} \Uparrow}\ \text{co}\right) = a : \mathit{Acts}(\mathcal{P}' \Uparrow)$$

Two programs that have the same action trace have the same observable input/output behavior. However, we would like to consider two programs that have action traces that differ only in the insertion and deletion of silent actions to have the same observable input/output behavior. Nonetheless, we do not wish to consider a terminating program with only silent actions to have the same observable behavior as a non-terminating program with only silent actions. This

motivates defining *action trace bisimilarity* $\mathcal{A}_1 \cong \mathcal{A}_2$ as follows:

$$\frac{}{\mathcal{A} \succeq \mathcal{A}} \qquad \frac{\mathcal{A} \succeq \mathcal{A}'}{\epsilon : \mathcal{A} \succeq \mathcal{A}'}$$

$$\frac{}{\bullet \simeq \bullet}\ \mathsf{CO} \qquad \frac{\mathcal{A}_1 \cong \mathcal{A}_2}{a : \mathcal{A}_1 \simeq a : \mathcal{A}_2}\ \mathsf{CO}$$

$$\frac{\mathcal{A}_1 \succeq \mathcal{A}_1' \qquad \mathcal{A}_2 \succeq \mathcal{A}_2' \qquad \mathcal{A}_1' \simeq \mathcal{A}_2'}{\mathcal{A}_1 \cong \mathcal{A}_2}\ \mathsf{CO}$$

$\mathcal{A} \succeq \mathcal{A}'$ is an inductively defined relation that holds when $\mathcal{A}'$ may be obtained from $\mathcal{A}$ by dropping an arbitrary, but finite, prefix of ϵ actions. $\mathcal{A}_1 \simeq \mathcal{A}_2$ and $\mathcal{A}_1 \cong \mathcal{A}_2$ are mutually, coinductively defined relations; the former holds when $\mathcal{A}_1$ and $\mathcal{A}_2$ are either both $\bullet$ or have equal heads and bisimilar tails, while the latter holds when $\mathcal{A}_1$ and $\mathcal{A}_2$ match after dropping ϵ-prefixes.

IO Fairness Our first notion of fairness, dubbed *IO fairness*, captures the behavior of a fair scheduler for the program's IO threads. Intuitively, a fair IO-thread scheduler ensures that every IO thread in the program makes progress. We formalize this intuitive notion as $\mathsf{IO\_FAIR}(\mathcal{P} \Uparrow)$:

$$\mathsf{IO\_FAIR}(\mathcal{P} \Uparrow) \stackrel{\text{def}}{=} \forall i \in \mathbb{N}.\ \forall \langle \theta, e_{io} \rangle \in I^{\mathcal{T}}(\mathcal{P} \Uparrow, i).\ \exists j > i.\ \langle \theta, e_{io} \rangle \notin I^{\mathcal{T}}(\mathcal{P} \Uparrow, j)$$

This predicate asserts that every IO thread in every thread soup in the program trace eventually "leaves" the thread soup; a thread $\langle \theta, e \rangle$ "leaves" the thread soup by either terminating (IOUnit), synchronizing and moving to the synchronization soup (IOSyncInit), or transitioning to a new IO expression (IOEval, IOFork, IOBindUnit, IOGetChar, and IOPutChar).[5] Note that in order to satisfy $\langle \theta, e_{io} \rangle \notin I^{\mathcal{T}}(\mathcal{P} \Uparrow, j)$, $I^{\mathcal{T}}(\mathcal{P} \Uparrow, j)$ must be defined.

Although we have not given a type system for transactional events in the present work, it is interesting to note that a program trace in which an IO thread exhibits a runtime type error (and "gets stuck") is necessarily unfair, since such an IO thread may never "leave" the thread soup.

Sync Fairness Our second notion of fairness, dubbed *sync fairness*, captures the behavior of a fair "synchronizer" for the program's synchronizing threads. A "synchronizer" is the mechanism by which the collection of synchronizing threads in the IOSyncSync rule are chosen. Intuitively, a fair "synchronizer" ensures that every synchronizing thread that could synchronize often enough does, in fact,

[5] We assume that sequential evaluation does not admit any expression e such that $e \hookrightarrow e$. This assumption implies that there is no program state $\mathcal{P}$ such that $\mathcal{P} \xrightarrow{a} \mathcal{P}$. This does not preclude expressions or program states that evaluate to themselves, but simply that such evaluations take more than one step.

synchronize. To capture the idea that a synchronizing thread could synchronize, we define $\mathsf{ENABLED}(\theta, \mathcal{S})$, a predicate asserting that θ is a sychronizing thread in the synchronization soup $\mathcal{S}$ that may synchronize with (zero or more) other synchronizing threads.

$$\begin{array}{l}\mathsf{ENABLED}(\theta, \mathcal{S}) \stackrel{\text{def}}{=} \\ \quad \exists\{\langle\theta, M^{IO}, e_{evt}\rangle, \langle\theta_1, M_1^{IO}, e_{evt1}\rangle, \ldots\} \subseteq \mathcal{S}. \\ \quad\quad \mathsf{SYNCABLE}(\{\langle\theta, e_{evt}\rangle, \langle\theta_1, e_{evt1}\rangle, \ldots\})\end{array}$$

Using enabledness, we formalize sync fairness as $\mathsf{SYNC\_FAIR}(\mathcal{P} \Uparrow)$:

$$\begin{array}{l}\mathsf{SYNC\_FAIR}(\mathcal{P} \Uparrow) \stackrel{\text{def}}{=} \\ \quad \forall i \in \mathbb{N}.\ \forall\langle\theta, M^{IO}, e_{evt}\rangle \in I^{\mathcal{S}}(\mathcal{P} \Uparrow, i). \\ \quad\quad \exists j > i.\ \langle\theta, M^{IO}, e_{evt}\rangle \notin I^{\mathcal{S}}(\mathcal{P} \Uparrow, j) \vee \\ \quad\quad\quad \forall k \geq j.\ \neg\mathsf{ENABLED}(\theta, I^{\mathcal{S}}(\mathcal{P} \Uparrow, k))\end{array}$$

This predicate asserts that every synchronizing thread in every synchronization soup in the program trace eventually either "leaves" the synchronization soup (and moves to the thread soup (IOSyncSync)) *or* is never again enabled. Since we described the behavior of a fair "synchronizer" as one that ensures that every synchronizing thread that is enabled often enough does synchronize, one may be confused by the appearance of the negation of the enabledness predicate in this definition. Consider sync unfairness, the negation of sync fairness:

$$\begin{array}{l}\mathsf{SYNC\_UNFAIR}(\mathcal{P} \Uparrow) \stackrel{\text{def}}{=} \neg\mathsf{SYNC\_FAIR}(\mathcal{P} \Uparrow) \equiv \\ \quad \exists i \in \mathbb{N}.\ \exists\langle\theta, M^{IO}, e_{evt}\rangle \in I^{\mathcal{S}}(\mathcal{P} \Uparrow, i). \\ \quad\quad \forall j > i.\ \langle\theta, M^{IO}, e_{evt}\rangle \in I^{\mathcal{S}}(\mathcal{P} \Uparrow, j) \wedge \\ \quad\quad\quad \exists k \geq j.\ \mathsf{ENABLED}(\theta, I^{\mathcal{S}}(\mathcal{P} \Uparrow, k))\end{array}$$

This predicate asserts that some synchronizing thread in some synchronization soup in the program trace always remains in the synchronization soup and eventually becomes enabled. Given an unfairly treated thread, we can witness an infinite number of indices ($k_1 < k_2 < k_3 < \cdots$) in which the thread could synchronize in the corresponding program states ($I^{\mathcal{S}}(\mathcal{P} \Uparrow, k_1)$, $I^{\mathcal{S}}(\mathcal{P} \Uparrow, k_2)$, $I^{\mathcal{S}}(\mathcal{P} \Uparrow, k_3)$, ...) by repeatedly instantiating $\forall j > i. \exists k \geq j. \mathsf{ENABLED}(\theta, I^{\mathcal{S}}(\mathcal{P} \Uparrow, k))$. Thus, a trace is sync unfair when it does not synchronize a synchronizing thread that is enabled infinitely often.[6]

[6] Our notion of sync fairness is, therefore, an instance of *strong fairness* [5, 2, 8]. We could define *weak sync fairness* as follows:

$$\begin{array}{l}\mathsf{WEAK\_SYNC\_FAIR}(\mathcal{P} \Uparrow) \stackrel{\text{def}}{=} \\ \quad \forall i \in \mathbb{N}.\ \forall\langle\theta, M^{IO}, e_{evt}\rangle \in I^{\mathcal{S}}(\mathcal{P} \Uparrow, i). \\ \quad\quad \exists j > i.\ \langle\theta, M^{IO}, e_{evt}\rangle \notin I^{\mathcal{S}}(\mathcal{P} \Uparrow, j) \vee \\ \quad\quad\quad \exists k \geq j.\ \neg\mathsf{ENABLED}(\theta, I^{\mathcal{S}}(\mathcal{P} \Uparrow, k))\end{array}$$

A trace is weakly sync unfair when it does not synchronize a synchronizing thread that is enabled continuously. Note that weak sync fairness admits an execution of the threads $\{\langle\theta_A, \texttt{forever (sync (recvEvt k))}\rangle, \langle\theta_B, \texttt{forever (sync (recvEvt k))}\rangle,$

Examples It is instructive to consider some examples that demonstrate what executions are admitted and required by IO and sync fairness.[7]

A classic example is a program with two threads that repeatedly send on a channel and a third thread that repeatedly receives on the channel:

$$\langle \theta_A, \texttt{forever (sync (sendEvt k 'a'))} \rangle \qquad \langle \theta_B, \texttt{forever (sync (sendEvt k 'b'))} \rangle$$
$$\langle \theta_C, \texttt{forever (sync (recvEvt k))} \rangle$$

IO and sync fairness demand that θ_C receive both `'a'`s and `'b'`s; that is, it would be unfair if, for example, θ_C only communicated with θ_A.

Another classic example is a program with two threads that repeatedly send on different channels and a third thread that repeatedly chooses between receiving on the two channels:

$$\langle \theta_A, \texttt{forever (sync (sendEvt ka 'a'))} \rangle \qquad \langle \theta_B, \texttt{forever (sync (sendEvt kb 'b'))} \rangle$$
$$\langle \theta_C, \texttt{forever (sync ((recvEvt ka) `chooseEvt` (recvEvt kb)))} \rangle$$

Again, IO and sync fairness demand that θ_C receive both `'a'`s and `'b'`s.

Note that in the previous example, the fact that a fair execution exercises both sub-events of the `chooseEvt` is due to the fact that the different sub-events are independently enabled by different threads. Consider this example, where one thread chooses between sending `'a'` and sending `'b'` to another thread:

$$\langle \theta_A, \texttt{forever (sync ((sendEvt k 'a') `chooseEvt` (sendEvt k 'b')))} \rangle$$
$$\langle \theta_B, \texttt{forever (sync (recvEvt k))} \rangle$$

IO and sync fairness admit executions of this program where θ_B receives only `'a'`s or only `'b'`s.

In this example, θ_A sends to θ_B, either directly or indirectly via θ_C:

$$\langle \theta_A, \texttt{forever (sync (sendEvt k 'a'))} \rangle \qquad \langle \theta_B, \texttt{forever (sync (recvEvt k))} \rangle$$
$$\langle \theta_C, \texttt{forever (sync ((recvEvt k) `thenEvt` (sendEvt k)))} \rangle$$

IO and sync fairness demand that thread θ_C repeatedly synchronizes, because, once thread θ_C is synchronizing, neither thread θ_A nor thread θ_B can synchronize until all three threads are synchronizing, in which case θ_C is enabled.

This final example is a program with five threads:

$$\langle \theta_A, \texttt{forever (sync (sendEvt k1 'a'))} \rangle \qquad \langle \theta_B, \texttt{forever (sync (recvEvt k1))} \rangle$$
$$\langle \theta_C, \texttt{forever (sync (sendEvt k2 'c'))} \rangle \qquad \langle \theta_D, \texttt{forever (sync (recvEvt k2))} \rangle$$
$$\langle \theta_E, \texttt{forever (sync ((sendEvt k1 'e') `thenEvt` (\ _ -> sendEvt k2 'e')))} \rangle$$

Perhaps counterintuitively, IO and sync fairness admit program executions of this program where thread θ_E synchronizes only a finite (including zero) number of times. Consider the program execution where (1) θ_E executes until it

$\langle \theta_3, \texttt{forever (sync (sendEvt k ()))} \rangle\}$ in which θ_C never synchronizes, since θ_A is not enabled whenever θ_C is not synchronizing. Weak sync fairness, therefore, seems to be of limited utility for reasoning about transactional events.

[7] For convenience, we use Haskell syntax.

is synchronizing, then (2) θ_A and θ_B execute until they are synchronizing and synchronize θ_A and θ_B, then (3) θ_C and θ_D execute until they are synchronizing and synchronize θ_C and θ_D, then repeat (2) and (3) infinitely. At no time is θ_E enabled, since θ_B and θ_D are never synchronizing at the same time.

5 Instrumented Semantics

The operational semantics of Section 3 and the fairness predicates of Section 4 provide a specification for fair transactional events, but do not immediately suggest an implementation. In this section, we instrument the original operational semantics and demonstrate that the instrumented operational semantics refines the original operational semantics in the sense that every IO-fair program trace in the instrumented semantics corresponds to an IO- and sync-fair program trace in the original semantics with the same observable input/output behavior.

Concurrent Evaluation ($\mathcal{P} \xrightarrow{a}_{\mathbf{inst}} \mathcal{P}'$) To motivate the instrumented operational semantics, we observe that if there is a program state where the IOSyncSync rule does not apply and a later program state where the IOSyncSync rule does apply, then there must be an occurrence of the IOSyncInit rule in the program trace that takes the former program state to the later. Hence, it is only necessary to check for the applicability of the IOSyncSync rule immediately after applications of the IOSyncInit rule. In fact, the instrumented operational semantics checks for the existence of a synchronizable group when evaluating an IO thread of the form $\langle \theta, M^{IO}[\texttt{sync}\ e_{evt}] \rangle$. If a synchronizable group exists, then the IO thread commits (with zero or more synchronizing threads) and continues as an IO thread (with its synchronization result); if no synchronizable group exists, then the IO thread blocks and transitions to a synchronizing thread. Thus, an IO thread has "one shot" to initiate its own synchronization.

The instrumented semantics also adds a *weight* to each synchronizing thread:

Weight	$w \in \mathbb{N}$
Synchronizing Thread	$S ::= \langle \theta, M^{IO}, e_{evt}, w \rangle$

Intuitively, the weight measures how long a synchronizing thread has been waiting to synchronize. When choosing a synchronization group to synchronize, the semantics selects a group with a synchronizing thread that has been waiting the longest (among all synchronizing threads that could synchronize).

We formalize the instrumented semantics by replacing the IOSyncInit and IOSyncSync rules of Fig. 3 with these IOSyncCommit and IOSyncBlock rules:

$$\frac{\begin{array}{l}\text{IOSyncCommit}\\ \{\langle\theta, e_{evt}\rangle, \langle\theta_1, e_{evt1}\rangle, \ldots\} \rightsquigarrow^* \{\langle\theta, \texttt{alwaysEvt}\ e'\rangle, \langle\theta_1, \texttt{alwaysEvt}\ e'_1\rangle, \ldots\} \\ \qquad W = \max\{0, w_1, \ldots\} \\ \quad \forall\{\langle\theta_a, M_a^{IO}, e_{evtz}, w_a\rangle, \ldots\} \subseteq \mathcal{S} \uplus \{\langle\theta_1, M_1^{IO}, e_1, w_1\rangle, \ldots\}. \\ \qquad \mathsf{SYNCABLE}(\{\langle\theta, e_{evt}\rangle, \langle\theta_a, e_{evtz}\rangle, \ldots\}) \Rightarrow W \geq \max\{0, w_a, \ldots\}\end{array}}{\begin{array}{l}\mathcal{T} \uplus \{\langle\theta, M^{IO}[\texttt{sync}\ e_{evt}]\rangle\}; \mathcal{S} \uplus \{\langle\theta_1, M_1^{IO}, e_{evt1}, w_1\rangle, \ldots\} \\ \quad \xrightarrow{\epsilon}_{\text{inst}} \mathcal{T} \uplus \{\langle\theta, M^{IO}[\texttt{unitIO}\ e']\rangle, \langle\theta_1, M_1^{IO}[\texttt{unitIO}\ e'_1]\rangle, \ldots\}; \mathit{incw}(\mathcal{S})\end{array}}$$

$$\frac{\begin{array}{l}\text{IOSyncBlock}\\ \neg\exists\{\langle\theta_1, M_1^{IO}, e_{evt1}, w_1\rangle, \ldots\} \subseteq \mathcal{S}. \\ \qquad \mathsf{SYNCABLE}(\{\langle\theta, e_{evt}\rangle, \langle\theta_1, e_{evt1}\rangle, \ldots\})\end{array}}{\mathcal{T} \uplus \{\langle\theta, M^{IO}[\texttt{sync}\ e_{evt}]\rangle\}; \mathcal{S} \xrightarrow{\epsilon}_{\text{inst}} \mathcal{T}; \mathcal{S} \uplus \{\langle\theta, M^{IO}, e_{evt}, 0\rangle\}}$$

The IOSyncCommit rule synchronizes one IO thread ($\langle\theta, M^{IO}[\texttt{sync}\ e_{evt}]\rangle$) along with a (possibly empty) set of synchronizing threads ($\{\langle\theta_1, M_1^{IO}, e_{evt1}, w_1\rangle, \ldots\}$). The weight of this group is $\max\{0, w_1, \ldots\}$; the 0 represents the weight of the IO thread and ensures that the group weight is defined if the set of synchronizing threads is empty. This group weight is required to be greater than or equal to the group weight of every other synchronizable group. All synchronizing threads that do not synchronize ($\mathcal{S}$) have their weights incremented ($\mathit{incw}(\mathcal{S})$), whether or not they could have synchronized along with the synchronizing IO thread. The IOSyncBlock rule transitions an IO thread that cannot synchronize with any existing synchronizing threads to a synchronizing thread with zero weight.

Traces, IO Fairness, and Sync Fairness We easily adapt the definitions of program traces and IO fairness from Section 4 as $\mathcal{P} \Uparrow_{\text{inst}}$ and $\mathsf{IO\_FAIR}_{\text{inst}}(\mathcal{P} \Uparrow_{\text{inst}})$.[8]

Adapting the definition of sync fairness requires a more substantial change. Since the instrumented semantics only adds a synchronizing thread to the synchronization soup if doing so does not create a synchronizable group, no synchronizing thread is ever ENABLED in any synchronization soup in a instrumented program trace. The instrumented semantics requires a different formalization of the idea that a synchronizing thread could synchronize; rather than being enabled at a particular program state in a program trace, a synchronizing thread is enabled by a particular IOSyncCommit step in a program trace:

$$\begin{array}{l}\mathsf{ENABLED}_{\text{inst}}(\theta, \mathcal{T}; \mathcal{S} \xrightarrow{a}_{\text{inst}} \mathcal{P}') \stackrel{\text{def}}{=} \\ \quad \exists\langle\theta_s, M_s^{IO}[\texttt{sync}\ e_{evts}]\rangle \in \mathcal{T}. \\ \qquad \mathcal{T}; \mathcal{S} \xrightarrow{a}_{\text{inst}} \mathcal{P}' \equiv \text{IOSyncCommit}(\langle\theta_s, M_s^{IO}[\texttt{sync}\ e_{evts}]\rangle) \wedge \\ \qquad \exists\{\langle\theta, M^{IO}, e_{evt}, w\rangle, \langle\theta_1, M_1^{IO}, e_{evt1}, w_1\rangle, \ldots\} \subseteq \mathcal{S}. \\ \qquad\quad \mathsf{SYNCABLE}(\{\langle\theta_s, e_{evts}\rangle, \langle\theta, e_{evt}\rangle, \langle\theta_1, e_{evt1}\rangle, \ldots\})\end{array}$$

[8] In order to avoid excessive notation, we overload the $I^{\rightarrow}$, $I^{\mathcal{P}}$, $I^{\mathcal{T}}$, $I^{\mathcal{S}}$, and *Acts* operations on instrumented program traces.

This predicate asserts that the program step $\mathcal{T}; \mathcal{S} \xrightarrow{a}_{\text{inst}} \mathcal{P}'$ is an instance of IOSyncCommit that commits the IO thread $\langle \theta_s, M_s^{IO}[\texttt{sync}\ e_{evts}] \rangle$ and that θ is a synchronizing thread in the synchronization soup $\mathcal{S}$ that may synchronize with the committing thread and with (zero or more) other synchronizing threads. Sync fairness for instrumented program traces is defined similarly to sync fairness for original program traces, but uses the $\mathsf{ENABLED}_{\text{inst}}$ predicate rather than the $\mathsf{ENABLED}$ predicate and does not allow a synchronizing thread to "leave" the synchronization soup by simply changing its weight:

$$\begin{array}{l}\mathsf{SYNC\_FAIR}_{\text{inst}}(\mathcal{P} \Uparrow_{\text{inst}}) \stackrel{\text{def}}{=} \\ \quad \forall i \in \mathbb{N}.\ \forall \langle \theta, M^{IO}, e_{evt}, w \rangle \in I^{\mathcal{S}}(\mathcal{P} \Uparrow_{\text{inst}}, i). \\ \quad\quad \exists j > i.\ \forall v.\ \langle \theta, M^{IO}, e_{evt}, v \rangle \notin I^{\mathcal{S}}(\mathcal{P} \Uparrow_{\text{inst}}, j) \vee \\ \quad\quad\quad \forall k > j.\ \neg \mathsf{ENABLED}_{\text{inst}}(\theta, \mathcal{I}^{\rightarrow}(\mathcal{P} \Uparrow_{\text{inst}}, k))\end{array}$$

Refines Original Semantics We show that the instrumented semantics refines the original semantics by demonstrating a translation from instrumented program traces to original program traces that preserves the properties of interest: observable input/output behavior, IO fairness, and sync fairness.

Fig. 4 gives the corecursively defined translation, using a simple erasure of weights from synchronizing threads. A IOSyncBlock step is translated to a IOSyncInit step; a IOSyncCommit step is translated to a IOSyncInit step followed by a IOSyncSync step; all other steps in the instrumented semantics are translated to the corresponding step in the original semantics. A terminal program state in the instrumented semantics is translated to a terminal program state in the original semantics. There is a subtlety that makes the translation partial: a program state such as $\{\}; \{\langle \theta, M^{IO}, \texttt{alwaysEvt}\ e', w \rangle\}$ is terminal in the instrumented semantics (all rules in the instrumented semantics require an IO thread), but is non-terminal in the original semantics (the IOSyncSync rule may step to the program state $\{\langle \theta, M^{IO}[\texttt{unitIO}\ e'] \rangle\}; \{\})$. However, if the initial program state in the instrumented program trace does not contain a syncable set of synchronizing threads, then terminal program states coincide (since the absence of syncable sets is preserved by the instrumented semantics (Lemma 1 of Appendix A)).

The following theorem establishes that every instrumented program trace that is IO and sync fair corresponds to an original program trace with the same observable input/output behavior that is IO and sync fair.

Theorem 1 (Instrumented semantics refines original semantics).
If $\mathcal{P} \Uparrow_{\text{inst}}$ *is a program trace in the instrumented semantics such that*

- $\forall \{\langle \theta_1, M_1^{IO}, e_{evt1}, w_1 \rangle, \ldots\} \subseteq I^{\mathcal{S}}(\mathcal{P} \Uparrow_{\text{inst}}, 0).\ \neg \mathsf{SYNCABLE}(\{\langle \theta_1, e_{evt1} \rangle, \ldots\})$,
- $\mathsf{IO\_FAIR}_{\text{inst}}(\mathcal{P} \Uparrow_{\text{inst}})$, *and*
- $\mathsf{SYNC\_FAIR}_{\text{inst}}(\mathcal{P} \Uparrow_{\text{inst}})$,

then

- $[\![\mathcal{P} \Uparrow_{\text{inst}}]\!]$ *is defined,*
- $\mathit{Acts}(\mathcal{P} \Uparrow_{\text{inst}}) \cong \mathit{Acts}([\![\mathcal{P} \Uparrow_{\text{inst}}]\!])$,

$$[\![\{\langle \theta, M^{IO}, e_{evt}, w\rangle, \ldots\}]\!] = \{\langle \theta, M^{IO}, e_{evt}\rangle, \ldots\}$$

$$[\![\mathcal{T}; \mathcal{S}]\!] = \mathcal{T}; [\![\mathcal{S}]\!]$$

$$\left[\!\!\left[\dfrac{\textsc{Term}\ \neg\exists a, \mathcal{P}'.\mathcal{P} \xrightarrow{a}_{\text{inst}} \mathcal{P}'}{\mathcal{P} \Uparrow_{\text{inst}}}\ \mathsf{co}\right]\!\!\right] = \dfrac{\textsc{Term}\ \left[\!\left[\neg\exists a, \mathcal{P}'.\mathcal{P} \xrightarrow{a}_{\text{inst}} \mathcal{P}'\right]\!\right]}{[\![\mathcal{P}]\!] \Uparrow}\ \mathsf{co}$$

$$\left[\!\!\left[\dfrac{\textsc{Step}\ \mathcal{P} \xrightarrow{a}_{\text{inst}} \mathcal{P}' \qquad \mathcal{P}' \Uparrow_{\text{inst}}}{\mathcal{P} \Uparrow_{\text{inst}}}\ \mathsf{co}\right]\!\!\right] =$$

$$\begin{cases}
\dfrac{\textsc{Step}\ [\![\mathcal{P}]\!] \xrightarrow{a} [\![\mathcal{P}']\!] \qquad [\![\mathcal{P}' \Uparrow_{\text{inst}}]\!]}{[\![\mathcal{P}]\!] \Uparrow}\ \mathsf{co} & \text{if } \mathcal{P} \xrightarrow{a}_{\text{inst}} \mathcal{P}' \notin \{\text{IOSyncCommit}, \text{IOSyncBlock}\} \\
\hline
\dfrac{\textsc{Step}\ [\![\mathcal{P}]\!] \xrightarrow{a} \mathcal{P}'' \qquad \dfrac{\textsc{Step}\ \mathcal{P}'' \xrightarrow{a''} [\![\mathcal{P}']\!] \qquad [\![\mathcal{P}' \Uparrow_{\text{inst}}]\!]}{\mathcal{P}'' \Uparrow}\ \mathsf{co}}{[\![\mathcal{P}]\!] \Uparrow}\ \mathsf{co} & \\
\quad \text{if } \mathcal{P} \xrightarrow{a}_{\text{inst}} \mathcal{P}' \equiv \dfrac{\begin{array}{c}\text{IOSyncCommit} \\ \{\langle \theta, e_{evt}\rangle, \langle \theta_1, e_{evt1}\rangle, \ldots\} \rightsquigarrow^* \{\langle \theta, \texttt{alwaysEvt}\ e'\rangle, \langle \theta_1, \texttt{alwaysEvt}\ e'_1\rangle, \ldots\} \\ W = \max\{0, w_1, \ldots\} \\ \forall\{\langle \theta_a, M_a^{IO}, e_{evtz}, w_a\rangle, \ldots\} \subseteq \mathcal{S} \uplus \{\langle \theta_1, M_1^{IO}, e_1, w_1\rangle, \ldots\}. \\ \mathsf{SYNCABLE}(\{\langle \theta, e_{evt}\rangle, \langle \theta_a, e_{evtz}\rangle, \ldots\}) \Rightarrow W \geq \max\{0, w_a, \ldots\}\end{array}}{\begin{array}{l}\mathcal{T} \uplus \{\langle \theta, M^{IO}[\texttt{sync}\ e_{evt}]\rangle\}; \mathcal{S} \uplus \{\langle \theta_1, M_1^{IO}, e_{evt1}, w_1\rangle, \ldots\} \\ \quad \xrightarrow{\epsilon}_{\text{inst}} \mathcal{T} \uplus \{\langle \theta, M^{IO}[\texttt{unitIO}\ e']\rangle, \langle \theta_1, M_1^{IO}[\texttt{unitIO}\ e'_1]\rangle, \ldots\}; incw(\mathcal{S})\end{array}} \\
\quad \text{where } [\![\mathcal{P}]\!] \xrightarrow{a} \mathcal{P}'' = \dfrac{\begin{array}{c}\text{IOSyncInit}\end{array}}{\begin{array}{l}\mathcal{T} \uplus \{\langle \theta, M^{IO}[\texttt{sync}\ e_{evt}]\rangle\}; [\![\mathcal{S}]\!] \uplus \{\langle \theta_1, M_1^{IO}, e_1\rangle, \ldots\} \\ \quad \xrightarrow{\epsilon}_{\text{inst}} \mathcal{T}; [\![\mathcal{S}]\!] \uplus \{\langle \theta, M^{IO}, e_{evt}\rangle, \langle \theta_1, M_1^{IO}, e_1\rangle, \ldots\}\end{array}} \\
\quad \text{and } \mathcal{P}'' \xrightarrow{a''} [\![\mathcal{P}']\!] = \dfrac{\begin{array}{c}\text{IOSyncSync} \\ \{\langle \theta, e\rangle, \langle \theta_1, e_1\rangle, \ldots\} \rightsquigarrow^* \{\langle \theta, \texttt{alwaysEvt}\ e'\rangle, \langle \theta_1, \texttt{alwaysEvt}\ e'_1\rangle, \ldots\}\end{array}}{\begin{array}{l}\mathcal{T}; [\![\mathcal{S}]\!] \uplus \{\langle \theta, M^{IO}, e_{evt}\rangle, \langle \theta_1, M_1^{IO}, e_{evt1}\rangle, \ldots\} \\ \quad \xrightarrow{\epsilon} \mathcal{T} \uplus \{\langle \theta, M^{IO}[\texttt{unitIO}\ e']\rangle, \langle \theta_1, M_1^{IO}[\texttt{unitIO}\ e'_1]\rangle, \ldots\}; [\![\mathcal{S}]\!]\end{array}} \\
\hline
\dfrac{\textsc{Step}\ [\![\mathcal{P}]\!] \xrightarrow{a} [\![\mathcal{P}']\!] \qquad [\![\mathcal{P}' \Uparrow_{\text{inst}}]\!]}{[\![\mathcal{P}]\!] \Uparrow}\ \mathsf{co} & \\
\quad \text{if } \mathcal{P} \xrightarrow{a}_{\text{inst}} \mathcal{P}' \equiv \dfrac{\begin{array}{c}\text{IOSyncBlock} \\ \neg\exists\{\langle \theta_1, M_1^{IO}, e_{evt1}, w_1\rangle, \ldots\} \subseteq \mathcal{S}. \\ \mathsf{SYNCABLE}(\{\langle \theta, e_{evt}\rangle, \langle \theta_1, e_{evt1}\rangle, \ldots\})\end{array}}{\mathcal{T} \uplus \{\langle \theta, M^{IO}[\texttt{sync}\ e_{evt}]\rangle\}; \mathcal{S} \xrightarrow{\epsilon}_{\text{inst}} \mathcal{T}; \mathcal{S} \uplus \{\langle \theta, M^{IO}, e_{evt}, 0\rangle\}} \\
\quad \text{where } [\![\mathcal{P}]\!] \xrightarrow{a} [\![\mathcal{P}']\!] = \dfrac{\text{IOSyncInit}}{\mathcal{T} \uplus \{\langle \theta, M^{IO}[\texttt{sync}\ e_{evt}]\rangle\}; [\![\mathcal{S}]\!] \xrightarrow{\epsilon} \mathcal{T}; [\![\mathcal{S}]\!] \uplus \{\langle \theta, M^{IO}, e_{evt}\rangle\}}
\end{cases}$$

Fig. 4. Translation of instrumented program traces to original program traces.

- $\mathsf{IO\_FAIR}([\![\mathcal{P} \Uparrow_{\mathrm{inst}}]\!])$, *and*
- $\mathsf{SYNC\_FAIR}([\![\mathcal{P} \Uparrow_{\mathrm{inst}}]\!])$.

The proof defines and uses a strictly monotonic function to map the index of a program state in $\mathcal{P} \Uparrow_{\mathrm{inst}}$ to the index of the equal program state in $[\![\mathcal{P} \Uparrow_{\mathrm{inst}}]\!]$; see Appendix A for details. Note that the proof does not make use of the weights of synchronizing threads.

This theorem does not establish that every original program trace that is IO and sync fair has a corresponding instrumented program trace with the same observable input/output behavior that is IO and sync fair. Indeed, the instrumented semantics necessarily excludes some fair original program traces. For example, consider three looping threads such that any pair of threads may synchronize together and all three threads may synchronize together; the original semantics includes traces in which the three threads synchronize, while the instrumented semantics includes only traces in which pairs of threads synchronize.

Guarantees Sync Fairness We defined sync fairness for instrumented program traces in order to simplify the proof of Theorem 1, using the sync fairness of an instrumented program trace to establish the sync fairness of the translated original program trace. Of course, the instrumentation of the instrumented semantics is meant to guide program executions towards sync fairness. The following theorem establishes that every instrumented program trace is sync fair, without additional assumptions.

Theorem 2 (Instrumented semantics is sync fair).
If $\mathcal{P} \Uparrow_{\mathrm{inst}}$ *is a program trace in the instrumented semantics, then* $\mathsf{SYNC\_FAIR}_{\mathrm{inst}}(\mathcal{P} \Uparrow_{\mathrm{inst}})$.

The proof is by contradiction and, intuitively, proceeds as follows. Suppose there is a sync-unfair instrumented program trace. Then there is a synchronizing thread that never "leaves" the synchronization soup, but is enabled at an infinite number of future program steps; call this the *unfair thread*. After the first of IOSyncCommit step at which the unfair thread is enabled but does not synchronize, the thread will necessarily have a weight greater than zero. Now consider the set of synchronizing threads that have weight greater than or equal to the unfair thread. This set can only decrease (Lemma 2), since new synchronizing threads enter the synchronization soup with weight zero and existing synchronizing threads that remain in the synchronization soup have their weights incremented together. Furthermore, this set must decrease at each IOSyncCommit step at which the unfair thread is enabled (Lemma 3), since the IOSyncCommit rule must commit an enabled synchronizing thread with weight greater than or equal to that of the unfair thread. Since there are an infinite number of future program steps at which the unfair thread is enabled, the set of synchronizing threads that have weight greater than or equal to the unfair thread must eventually be exhausted. At this point, the unfair thread must have weight greater than that of any other enabled synchronizing threads and must commit.

But, this contradicts the assumption that the thread never "leaves" the synchronization soup. Thus, there cannot be a sync-unfair instrumented program trace. See Appendix A for additional details and supporting lemmas.Together, Theorems 1 and 2 demonstrate that any program trace in the instrumented semantics corresponds to a sync-fair program trace in the original semantics.

6 Implementation

The obvious difficulty with implementing the instrumented semantics of Section 5 is knowing when to apply the IOSyncCommit and IOSyncBlock rules and which of the two rules to apply. While the two rules are mutually exclusive, they effectively require enumerating all of the enabled synchronization groups. Unfortunately, for the general case of transactional events, the enabledness of a set of event synchronizations is undecidable. To see this, we simply recall that sequential evalution can express recursively defined `Evt` and `IO` computations. Thus, an $\mathcal{E} \rightsquigarrow^*$ evaluation may diverge by being either infinitely deep (recursing through `thenEvt`) or infinitely wide (recursing through `chooseEvt`).

We therefore introduce *decidable synchronization groups*. A synchronization group $\mathcal{E}$ is a decidable synchronization group if all evaluations of $\mathcal{E} \rightsquigarrow^*$ terminate. That is, for all events in the synchronization group, all "paths" through the event are finite. Note that this is a property of a synchronization group, not an event, since an event synchronization may have control-flow that is based on values received from other events synchronizing in the group.

An implementation of fair transactional events, under the assumption of decidable synchronization groups, works as follows. When the fair IO thread scheduler selects a thread performing a synchronization, its event expression is evaluated with the blocked event expressions in the synchronization soup. If enabled synchronization groups emerge, then the IOSyncCommit rule is taken, choosing the synchronization group with maximum weight. If no enabled synchronization groups emerge, then the IOSyncBlock rule is taken. Note that the implementation may utilize many of the techniques described in the previous (non-fair) implementation of transactional events [4]. In particular, search threads and channel stores may be used to represent the synchronization soup in a manner that keeps all potential synchronizations evaluated "as much as possible" so that all work performed upon selecting a thread for synchronization is "new" work.

To represent thread weights, it suffices to use a global counter that records the number of IOSyncCommit steps taken. When a IOSyncBlock step is taken, the newly synchronizing thread records the current value of the global counter. Thus, the thread with maximum weight is simply the thread with the minimum recorded value.

Note that this suggested implementation will fail in the presence of nonterminating synchronization computations. In particular, if it is not possible to decide which of IOSyncCommit or IOSyncBlock applies to a particular thread synchronization, then no subsequent thread synchronizations can be initiated.

We conjecture that it is impossible to construct an implementation which enforces fairness in the presence of undecidable synchronization computations, since sync fairness depends upon knowing the enabledness of these computations. Verifying that all synchronization groups that arise during any execution of a program are decidable synchronization groups can be challenging, but appears to be reasonable in practice. In fact, all of the examples from prior work (e.g., guarded receive, Concurrent ML encoding, three-way rendezvous, boolean satisfiability encoding) [3, 4] give rise to only decidable synchronization groups. A notion similar to decidable synchronization groups, that of "obstruction freedom", is often employed in the context of Software Transactional Memory (STM). Obstruction-freedom asserts that a transaction executed in isolation commits in a finite number of steps. Again, this condition can be challenging to reason about in the presence of first-class transactions, and so the criticism applies equally well to STM systems such as STM Haskell [6].

The "global lock" approach, where there is exactly one thread attempting a synchronization at any time, might be relaxed by permitting multiple threads to begin searching for a synchronization, but tracking the order in which they began and requiring that they commit or block in that order. Alternatively, an implementation might track weights but permit unrestricted synchronization until a blocked thread's weight grows too large, after which point the global lock is enforced until no threads have weights above the threshold.

7 Related Work

Transactional events draws inspiration from both Concurrent ML [12] and Software Transactional Memory [14, 6].

Concurrent ML Reppy discusses fairness for Concurrent ML (CML) [12, Appendix B], but does not completely formalize the definition and there is no proof that the implementation of CML is fair. One notable difference is that CML discusses enabledness of synchronization objects (channels and conditions) in addition to enabledness of threads. Consider the following program:

$$\langle \theta_A, \texttt{forever (sync ((sendEvt k1 'a') `chooseEvt` (sendEvt k2 'b')))} \rangle$$
$$\langle \theta_B, \texttt{forever (sync ((recvEvt k1) `chooseEvt` (recvEvt k2)))} \rangle$$

CML's notion of fairness demands that thread θ_B receive both `'a'`s and `'b'`s (since channels `k1` and `k2` are both enabled infinitely often), while our notion of fairness allows thread θ_B to receive only `'a'`s or `'b'`s. As another example, consider the following program:

$$\left\langle \theta_A, \begin{array}{l}\texttt{let evt = (alwaysEvt 'a') `chooseEvt` (alwaysEvt 'b') in}\\ \texttt{forever (sync evt)}\end{array} \right\rangle$$

CML adopts a call-by-value evaluation strategy and the evaluation of an `alwaysEvt` allocates a fresh condition; hence, in the program above, the thread

repeatedly synchronizes on the same conditions and the synchronization must yield both `'a`'s and `'b`'s. Our semantics for transactional events treats `alwaysEvt` as a pure expression and allows the synchronization to yield only `'a`'s or `'b`'s.

The implementation of CML enforces fairness through a combination of channel priorities and an ordered queue of threads blocked on a channel. The choice of synchronization among a collection of enabled events with equal priority is resolved by a pseudo-random number generator; hence, the implementation of CML provides only a probabilistic guarantee of fairness.

Software Transactional Memory While it seems reasonable to propose a notion of fairness for Software Transactional Memory (STM) [14] similar in spirit to those proposed for CML and transactional events, the discussion is often informal or absent. For instance, STM Haskell [6] offers no discussion of fairness and the semantics allows a transaction that could commit to remain blocked indefinitely. Most work on STM has focused on the use of *contention managers* [7, 13, 15] that use heuristics to increase the throughput of an STM implementation (and behave "fair enough"), but without providing a guarantee of (complete) fairness. Various STM implementations are shown to be non-blocking (wait-free, lock-free, obstruction-free), although it is not always clear whether the property applies to the low-level implementation (threads attempting transactions make progress, but progress includes observing a conflict and aborting) or the high-level semantics (threads attempting transactions make progress and commit). Furthermore, properties such as obstruction freedom, which assert that a transaction completes so long as no other transactions are being attempted, are not directly applicable to transactional events. Any thread performing a synchronization that involves a send or receive necessarily requires another thread to be performing a synchronization with the matching communication, so such isolation properties are not appropriate.

8 Conclusion

We have formally characterized a notion of fairness in Transactional Events, a combination of first-class synchronous message-passing and all-or-nothing transactions. Our fairness notion permits programmers to reason with the guarantee that a synchronizing thread will not be blocked indefinitely from synchronizing by the mechanism which chooses synchronizations. We have given an instrumented operational semantics for transactional events along with theorems establishing that all program executions in this semantics are fair. Finally, we have described a condition on event synchronizations that permits an implementation of this semantics to successfully complete enabled synchronizations

Acknowledgments This work was supported in part by an RIT Center for Student Innovation Undergraduate Summer Research and Innovation Fellowship.

References

1. Amsden, E., Fluet, M.: Fairness for transactional events. In: Implementation and Application of Functional Languages: 23rd International Symposium (IFL'11). Lecture Notes in Computer Science, Springer Berlin / Heidelberg (2012), to appear
2. Costa, G., Stirling, C.: Weak and strong fairness in CCS. Information and Computation 73(3), 207–244 (1987)
3. Donnelly, K., Fluet, M.: Transactional events. In: Proceedings of the Eleventh ACM SIGPLAN International Conference on Functional Programming (ICFP'06). pp. 124–135. ACM Press (2006)
4. Donnelly, K., Fluet, M.: Transactional events. The Journal of Functional Programming 18(5–6), 649–706 (2008)
5. Francez, N.: Fairness. Texts and Monographs in Computer Science, Springer-Verlag, New York, NY, USA (1986)
6. Harris, T., Marlow, S., Peyton Jones, S., Herlihy, M.: Composable memory transactions. In: Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'05). pp. 48–60. ACM Press (2005)
7. Herlihy, M., Luchangco, V., Moir, M., Scherer, III, W.N.: Software transactional memory for dynamic-sized data structures. In: Proceedings of the 22nd Annual Symposium on Principles of Distributed Computing (PODC'03). pp. 92–101. ACM Press (2003)
8. Kwiatkowska, M.: Survey of fairness notions. Information and Software Technology 31(7), 371–386 (1989)
9. Peyton Jones, S.: Tackling the awkward squad: Monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell. In: Hoare, T., Broy, M., Steinbrüggen, R. (eds.) Engineering Theories of Software Construction, NATO Science Series: Computer & Systems Sciences, vol. 180, pp. 47–96. IOS Press, Amsterdam, The Netherlands (2001)
10. Peyton Jones, S., Gordon, A., Finne, S.: Concurrent Haskell. In: Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'96). pp. 295–308. ACM Press (1996)
11. Peyton Jones, S., Wadler, P.: Imperative functional programming. In: Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'93). pp. 71–84. ACM Press (1993)
12. Reppy, J.: Concurrent Programming in ML. Cambridge University Press, Cambridge, UK (1999)
13. Scherer, III, W.N., Scott, M.L.: Advanced contention management for dynamic software transactional memory. In: Proceedings of the 24th Annual Symposium on Principles of Distributed Computing (PODC'05). pp. 240–248. ACM Press (2005)
14. Shavit, N., Touitou, D.: Software transactional memory. Distributed Computing 10(2), 99–116 (1997)
15. Spear, M.F., Dalessandro, L., Marathe, V.J., Scott, M.L.: A comprehensive strategy for contention management in software transactional memory. In: Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'05). pp. 141–150. ACM Press (2009)

A Instrumented Semantics

Refines Original Semantics This appendix provides a proof sketch for Theorem 1, which establishes that every instrumented program trace that is IO and sync fair corresponds to an original program trace with the same observable input/output behavior that is IO and sync fair.

Lemma 1 (Instrumented semantics preserves absence of syncable groups).
If $\mathcal{T};\mathcal{S} \xrightarrow{a}_{\text{inst}} \mathcal{T}';\mathcal{S}'$ *is a step in the one-shot semantics such that*
- $\forall\{\langle\theta_1, M_1^{IO}, e_{evt1}, w_1\rangle, \ldots\} \subseteq \mathcal{S}.\ \neg\mathsf{SYNCABLE}(\{\langle\theta_1, e_{evt1}\rangle, \ldots\})$,

then
- $\forall\{\langle\theta_1, M_1^{IO}, e_{evt1}, w_1\rangle, \ldots\} \subseteq \mathcal{S}'.\ \neg\mathsf{SYNCABLE}(\{\langle\theta_1, e_{evt1}\rangle, \ldots\})$.

Proof sketch.
By inspection of the concurrent evaluation rules for the instrumented semantics.

Theorem 1 (Instrumented semantics refines original semantics).
If $\mathcal{P} \Uparrow_{\text{inst}}$ *is a program trace in the one-shot semantics such that*
- $\forall\{\langle\theta_1, M_1^{IO}, e_{evt1}, w_1\rangle, \ldots\} \subseteq I^{\mathcal{S}}(\mathcal{P} \Uparrow_{\text{inst}}, 0).\ \neg\mathsf{SYNCABLE}(\{\langle\theta_1, e_{evt1}\rangle, \ldots\})$,
- $\mathsf{IO\_FAIR}_{\text{inst}}(\mathcal{P} \Uparrow_{\text{inst}})$, *and*
- $\mathsf{SYNC\_FAIR}_{\text{inst}}(\mathcal{P} \Uparrow_{\text{inst}})$,

then
- $[\![\mathcal{P} \Uparrow_{\text{inst}}]\!]$ *is defined,*
- $\mathit{Actions}(\mathcal{P} \Uparrow_{\text{inst}}) \cong \mathit{Actions}([\![\mathcal{P} \Uparrow_{\text{inst}}]\!])$,
- $\mathsf{IO\_FAIR}([\![\mathcal{P} \Uparrow_{\text{inst}}]\!])$, *and*
- $\mathsf{SYNC\_FAIR}([\![\mathcal{P} \Uparrow_{\text{inst}}]\!])$.

Proof sketch.
- $[\![\mathcal{P} \Uparrow_{\text{inst}}]\!]$ is defined:

By coinduction on $\mathcal{P} \Uparrow_{\text{inst}}$, using $\forall\{\langle\theta_1, M_1^{IO}, e_{evt1}, w_1\rangle, \ldots\} \subseteq I^{\mathcal{S}}(\mathcal{P} \Uparrow_{\text{inst}}, 0).\ \neg\mathsf{SYNCABLE}(\{\langle\theta_1, e_{evt1}\rangle, \ldots\})$ and Lemma 1.
- $\mathit{Acts}(\mathcal{P} \Uparrow_{\text{inst}}) \cong \mathit{Acts}([\![\mathcal{P} \Uparrow_{\text{inst}}]\!])$:

By coinduction on $\mathcal{P} \Uparrow_{\text{inst}}$. For a terminal program state, we must show $\bullet \cong \bullet$, which follows by taking $\bullet \succeq \bullet$ and $\bullet \succeq \bullet$ and showing $\bullet \simeq \bullet$. For all steps other than IOSyncCommit, we must show $a{:}\mathit{Acts}(\mathcal{P}' \Uparrow_{\text{inst}}) \cong a{:}\mathit{Acts}([\![\mathcal{P}' \Uparrow_{\text{inst}}]\!])$ under the assumption $\mathit{Acts}(\mathcal{P}' \Uparrow_{\text{inst}}) \cong \mathit{Acts}([\![\mathcal{P}' \Uparrow_{\text{inst}}]\!])$, which follows by taking $a{:}\mathit{Acts}(\mathcal{P}' \Uparrow_{\text{inst}}) \succeq a{:}\mathit{Acts}(\mathcal{P}' \Uparrow_{\text{inst}})$ and $a{:}\mathit{Acts}([\![\mathcal{P}' \Uparrow_{\text{inst}}]\!]) \succeq a{:}\mathit{Acts}([\![\mathcal{P}' \Uparrow_{\text{inst}}]\!])$ and showing $a{:}\mathit{Acts}(\mathcal{P}' \Uparrow_{\text{inst}}) \simeq a{:}\mathit{Acts}([\![\mathcal{P}' \Uparrow_{\text{inst}}]\!])$. For an IOSyncCommit step, we must show $\epsilon{:}\mathit{Acts}(\mathcal{P}' \Uparrow_{\text{inst}}) \cong \epsilon{:}\epsilon{:}\mathit{Acts}([\![\mathcal{P}' \Uparrow_{\text{inst}}]\!])$ under the assumption $\mathit{Acts}(\mathcal{P}' \Uparrow_{\text{inst}}) \cong \mathit{Acts}([\![\mathcal{P}' \Uparrow_{\text{inst}}]\!])$, which follows by taking $\epsilon{:}\mathit{Acts}(\mathcal{P}' \Uparrow_{\text{inst}}) \succeq \epsilon{:}\mathit{Acts}(\mathcal{P}' \Uparrow_{\text{inst}})$ and $\epsilon{:}\epsilon{:}\mathit{Acts}([\![\mathcal{P}' \Uparrow_{\text{inst}}]\!]) \succeq \epsilon{:}\mathit{Acts}([\![\mathcal{P}' \Uparrow_{\text{inst}}]\!])$ and showing $\epsilon{:}\mathit{Acts}(\mathcal{P}' \Uparrow_{\text{inst}}) \simeq \epsilon{:}\mathit{Acts}([\![\mathcal{P}' \Uparrow_{\text{inst}}]\!])$.
- $\mathsf{IO\_FAIR}([\![\mathcal{P} \Uparrow_{\text{inst}}]\!])$:

Define, by induction, a strictly monotonic function f to map the index of a program state in $\mathcal{P} \Uparrow_{\text{inst}}$ to the index of the equal program state in $[\![\mathcal{P} \Uparrow_{\text{inst}}]\!]$. The

inverse of this function f^{-1} maps the index of a program state in $[\![\mathcal{P} \Uparrow_{\text{inst}}]\!]$ to the index of the equal program state in $\mathcal{P} \Uparrow_{\text{inst}}$, but is undefined on indices that correspond to the intermediate program state in the translation of the IOSyncCommit rule. However, note that for all $i \in \mathbb{N}$ such that $f^{-1}(i)$ is undefined, $f^{-1}(i+1)$ is defined and $I^{\mathcal{T}}([\![\mathcal{P} \Uparrow_{\text{inst}}]\!], i) \subseteq I^{\mathcal{T}}([\![\mathcal{P} \Uparrow_{\text{inst}}]\!], i+1)$. Thus, given $i \in \mathbb{N}$ and $\langle \theta, e_{io} \rangle \in I^{\mathcal{T}}([\![\mathcal{P} \Uparrow_{\text{inst}}]\!], i)$, instantiate $\mathsf{IO\_FAIR}_{\text{inst}}(\mathcal{P} \Uparrow_{\text{inst}})$ with $i' = f^{-1}(i)$ (if $f^{-1}(i)$ is defined) or with $i' = f^{-1}(i+1)$ (if $f^{-1}(i)$ is undefined) and with $\langle \theta, e_{io} \rangle \in I^{\mathcal{T}}(\mathcal{P} \Uparrow_{\text{inst}}, i')$, obtain $j' > i'$ such that $\langle \theta, e_{io} \rangle \notin I^{\mathcal{T}}(\mathcal{P} \Uparrow_{\text{inst}}, j')$, and witness $j = f(j')$; $j > i$ follows from $j' > i'$ and the strict monotonicity of f and $\langle \theta, e_{io} \rangle \notin I^{\mathcal{T}}([\![\mathcal{P} \Uparrow_{\text{inst}}]\!], j)$ follows from $\langle \theta, e_{io} \rangle \notin I^{\mathcal{T}}(\mathcal{P} \Uparrow_{\text{inst}}, j')$.

- $\mathsf{SYNC\_FAIR}([\![\mathcal{P} \Uparrow_{\text{inst}}]\!])$:

Recall the function f defined above. Note that for all $i \in \mathbb{N}$ such that $f^{-1}(i)$ is undefined, $f^{-1}(i-1)$ and $f^{-1}(i+1)$ are defined, $f^{-1}(i-1)+1 = f^{-1}(i+1)$, $I^{\rightarrow}(\mathcal{P} \Uparrow_{\text{inst}}, f^{-1}(i-1)) \equiv \text{IOSyncCommit}(\langle \theta_s, M_s^{IO}[\texttt{sync}\ e_{evts}] \rangle)$, $I^{\mathcal{S}}([\![\mathcal{P} \Uparrow_{\text{inst}}]\!], i-1) \uplus \{\langle \theta_s, M_s^{IO}, e_{evts} \rangle\} = I^{\mathcal{S}}([\![\mathcal{P} \Uparrow_{\text{inst}}]\!], i)$, and $I^{\mathcal{S}}([\![\mathcal{P} \Uparrow_{\text{inst}}]\!], i+1) \subseteq I^{\mathcal{S}}([\![\mathcal{P} \Uparrow_{\text{inst}}]\!], i-1)$. Given $i \in \mathbb{N}$ and $\langle \theta, M^{IO}, e_{evt} \rangle \in I^{\mathcal{S}}([\![\mathcal{P} \Uparrow_{\text{inst}}]\!], i)$, if $f^{-1}(i)$ is undefined and $I^{\rightarrow}(\mathcal{P} \Uparrow_{\text{inst}}, f^{-1}(i-1)) \equiv \text{IOSyncCommit}(\langle \theta, M^{IO}[\texttt{sync}\ e_{evt}] \rangle)$ and $\forall z.\ \langle \theta, M^{IO}, e_{evt}, z \rangle \notin I^{\mathcal{S}}(\mathcal{P} \Uparrow_{\text{inst}}, f^{-1}(i+1))$, then witness $j = i+1$; $\langle \theta, M^{IO}, e_{evt} \rangle \notin I^{\mathcal{S}}([\![\mathcal{P} \Uparrow_{\text{inst}}]\!], j)$ follows from $\forall z.\ \langle \theta, M^{IO}, e_{evt}, z \rangle \notin I^{\mathcal{S}}(\mathcal{P} \Uparrow_{\text{inst}}, f^{-1}(i+1))$. Otherwise, given $i \in \mathbb{N}$ and $\langle \theta, M^{IO}, e_{evt} \rangle \in I^{\mathcal{S}}([\![\mathcal{P} \Uparrow_{\text{inst}}]\!], i)$, instantiate $\mathsf{SYNC\_FAIR}_{\text{inst}}(\mathcal{P} \Uparrow_{\text{inst}})$ with $i' = f^{-1}(i)$ (if $f^{-1}(i)$ is defined, noting $\exists w.\ \langle \theta, M^{IO}, e_{evt}, w \rangle \in I^{\mathcal{S}}(\mathcal{P} \Uparrow_{\text{inst}}, f^{-1}(i))$ follows from $\langle \theta, M^{IO}, e_{evt} \rangle \in I^{\mathcal{S}}([\![\mathcal{P} \Uparrow_{\text{inst}}]\!], i)$) or with $i' = f^{-1}(i-1)$ (if $f^{-1}(i)$ is undefined and $I^{\rightarrow}(\mathcal{P} \Uparrow_{\text{inst}}, f^{-1}(i-1)) \equiv \text{IOSyncCommit}(\langle \theta, M^{IO}[\texttt{sync}\ e_{evt}] \rangle)$ and $\exists z.\ \langle \theta, M^{IO}, e_{evt}, z \rangle \in I^{\mathcal{S}}(\mathcal{P} \Uparrow_{\text{inst}}, f^{-1}(i+1))$, noting $\exists w.\ \langle \theta, M^{IO}, e_{evt}, w \rangle \in I^{\mathcal{S}}(\mathcal{P} \Uparrow_{\text{inst}}, f^{-1}(i-1))$ follows from $\langle \theta, M^{IO}, e_{evt} \rangle \in I^{\mathcal{S}}([\![\mathcal{P} \Uparrow_{\text{inst}}]\!], i)$, $I^{\mathcal{S}}([\![\mathcal{P} \Uparrow_{\text{inst}}]\!], i-1) \uplus \{\langle \theta_s, M_s^{IO}, e_{evts} \rangle\} = I^{\mathcal{S}}([\![\mathcal{P} \Uparrow_{\text{inst}}]\!], i)$, $I^{\mathcal{S}}([\![\mathcal{P} \Uparrow_{\text{inst}}]\!], i+1) \subseteq I^{\mathcal{S}}([\![\mathcal{P} \Uparrow_{\text{inst}}]\!], i-1)$, and $\exists z.\ \langle \theta, M^{IO}, e_{evt}, z \rangle \in I^{\mathcal{S}}(\mathcal{P} \Uparrow_{\text{inst}}, f^{-1}(i+1))$) and with $\langle \theta, M^{IO}, e_{evt}, w \rangle \in I^{\mathcal{S}}(\mathcal{P} \Uparrow_{\text{inst}}, i')$, obtain $j' > i'$ such that either $\forall v.\ \langle \theta, M^{IO}, e_{evt}, v \rangle \notin I^{\mathcal{S}}(\mathcal{P} \Uparrow_{\text{inst}}, j')$ or $\forall k' \geq j'.\ \neg\mathsf{ENABLED}_{\text{inst}}(\theta, I^{\rightarrow}(\mathcal{P} \Uparrow_{\text{inst}}, k'))$. If $\forall v.\ \langle \theta, M^{IO}, e_{evt}, v \rangle \notin I^{\mathcal{S}}(\mathcal{P} \Uparrow_{\text{inst}}, j')$, then witness $j = f(j')$; $j > i$ follows from $j' > i'$ and the strict monotonicity of f (and $f^{-1}(i-1)+1 = f^{-1}(i+1)$ if $i' = f^{-1}(i-1)$) and $\langle \theta, M^{IO}, e_{evt} \rangle \notin I^{\mathcal{S}}([\![\mathcal{P} \Uparrow_{\text{inst}}]\!], j)$ follows from $\forall v.\ \langle \theta, M^{IO}, e_{evt}, v \rangle \notin I^{\mathcal{S}}(\mathcal{P} \Uparrow_{\text{inst}}, j')$. Otherwise, assume $\forall k' \geq j'.\ \neg\mathsf{ENABLED}_{\text{inst}}(\theta, I^{\rightarrow}(\mathcal{P} \Uparrow_{\text{inst}}, k'))$ and witness $j = f(j')+1$; $j > i$ follows from $j' > i'$ and the strict monotonicity of f (and $f^{-1}(i-1)+1 = f^{-1}(i+1)$ if $i' = f^{-1}(i-1)$) and show $\forall k \geq j.\ \neg\mathsf{ENABLED}(\theta, I^{\mathcal{S}}([\![\mathcal{P} \Uparrow_{\text{inst}}]\!], k))$. Given $k \geq j$, if $f^{-1}(k)$ is defined, then $\neg\mathsf{ENABLED}(\theta, I^{\mathcal{S}}([\![\mathcal{P} \Uparrow_{\text{inst}}]\!], k))$ follows from $\forall \{\langle \theta_1, M_1^{IO}, e_{evt1}, w_1 \rangle, \ldots\} \subseteq I^{\mathcal{S}}(\mathcal{P} \Uparrow_{\text{inst}}, f^{-1}(k))$. $\neg\mathsf{SYNCABLE}(\{\langle \theta_1, e_{evt1} \rangle, \ldots\})$, which follows from $\forall \{\langle \theta_1, M_1^{IO}, e_{evt1,w_1} \rangle, \ldots\} \subseteq I^{\mathcal{S}}(\mathcal{P} \Uparrow_{\text{inst}}, 0)$. $\neg\mathsf{SYNCABLE}(\{\langle \theta_1, e_{evt1} \rangle, \ldots\})$ and Lemma 1. Otherwise, given $k \geq j$ and $f^{-1}(k)$ is

undefined, $\neg\mathsf{ENABLED}(\theta, I^{\mathcal{S}}([\![\mathcal{P} \Uparrow_{\text{inst}}]\!], k))$ follows from $\neg\mathsf{ENABLED}_{\text{inst}}(\theta, I^{\rightarrow}(\mathcal{P} \Uparrow_{\text{inst}}, f^{-1}(k-1)))$, which follows from $\forall k' \geq j'.\ \neg\mathsf{ENABLED}_{\text{inst}}(\theta, I^{\rightarrow}(\mathcal{P} \Uparrow_{\text{inst}}, k'))$ and $f^{-1}(k-1) \geq j'$.

Guarantees Sync Fairness This appendix provides a proof sketch for Theorem 2, which establishes that every instrumented program trace is sync fair, without additional assumptions.

The "competitors" of a synchronizing thread in the synchronization soup is the set of synchronizing threads (including the thread under consideration) with weight equal to or greater than the weight of the thread under consideration. We formalize this as $\mathsf{COMPETITORS}(\theta_z, \mathcal{S})$:

$$\mathsf{COMPETITORS}(\theta_z, \mathcal{S}) \stackrel{\text{def}}{=} \begin{cases} \{\langle \theta, M^{IO}, e_{evt}, w\rangle \in \mathcal{S} \mid w \geq w_z\} & \text{if } \langle \theta_z, M_z^{IO}, e_{evtz}, w_z\rangle \in \mathcal{S} \\ \bot & \text{otherwise} \end{cases}$$

Lemma 2 states that, for a synchronizing thread with weight greater than 0 in a program state, at the next program state, it has either participated in a commit (and left the synchronization soup) or its set of competitors has not increased.

Lemma 2.
For all $\mathcal{P} \Uparrow_{\text{inst}}$, for all $i \in \mathbb{N}$, and for all $\langle \theta, M^{IO}, e_{evt}, w\rangle \in I^{\mathcal{S}}(\mathcal{P} \Uparrow_{\text{inst}}, i)$, if $w > 0$, then either:

- $\forall v.\ \langle \theta, M^{IO}, e_{evt}, v\rangle \notin I^{\mathcal{S}}(\mathcal{P} \Uparrow_{\text{inst}}, i+1)$, *or*
- $\mathsf{COMPETITORS}(\theta, I^{\mathcal{S}}(\mathcal{P} \Uparrow_{\text{inst}}, i+1)) \subseteq \mathsf{COMPETITORS}(\theta, I^{\mathcal{S}}(\mathcal{P} \Uparrow_{\text{inst}}, i))$.

Lemma 2 states that, for a synchronizing thread that is enabled in a program state, at the next program state, it has either participated in the commit (and left the synchronization soup) or its set of competitors has strictly decreased.

Lemma 3.
For all $\mathcal{P} \Uparrow_{\text{inst}}$, for all $i \in \mathbb{N}$, and for all $\langle \theta, M^{IO}, e, w\rangle \in I^{\mathcal{S}}(\mathcal{P} \Uparrow_{\text{inst}}, i)$, if $\mathsf{ENABLED}_{\text{inst}}(\theta, I^{\rightarrow}(\mathcal{P} \Uparrow_{\text{inst}}, i))$, then either:

- $\forall v.\ \langle \theta, M^{IO}, e_{evt}, v\rangle \notin I^{\mathcal{S}}(\mathcal{P} \Uparrow_{\text{inst}}, i+1)$, *or*
- $\mathsf{COMPETITORS}(\theta, I^{\mathcal{S}}(\mathcal{P} \Uparrow_{\text{inst}}, i+1)) \subset \mathsf{COMPETITORS}(\theta, I^{\mathcal{S}}(\mathcal{P} \Uparrow_{\text{inst}}, i))$

Lemma 4 states that any thread which is enabled infinitely often in an instrumented trace must, at some point, participate in a commit and leave the synchronizing soup.

Lemma 4.
For all $\mathcal{P} \Uparrow_{\text{inst}}$, for all $i \in \mathbb{N}$, and for all $\langle \theta, M^{IO}, e_{evt}, w\rangle \in I^{\mathcal{S}}(\mathcal{P} \Uparrow_{\text{inst}}, i)$, if $w > 0$ and $\forall j > i.\ \exists k \geq j.\ \mathsf{ENABLED}(\theta, I^{\rightarrow}(\mathcal{P} \Uparrow_{\text{inst}}, k))$, then $\exists m > i.\ \forall u.\ \langle \theta, M^{IO}, e_{evt}, u\rangle \notin I^{\mathcal{S}}(\mathcal{P} \Uparrow_{\text{inst}}, m)$.

Theorem 2 is the proof of sync fairness for the system. It demonstrates that any trace in the instrumented semantics is a fair trace.

Theorem 2 (Instrumented semantics is sync fair).
If $\mathcal{P} \Uparrow_{\text{inst}}$ *is a program trace in the instrumented semantics, then* $\mathsf{SYNC\_FAIR}_{\text{inst}}(\mathcal{P} \Uparrow_{\text{inst}})$.

Proof.
By contradiction.
Given $\mathcal{P} \Uparrow_{\text{inst}}$, assume $\neg\mathsf{SYNC\_FAIR}_{\text{inst}}(\mathcal{P} \Uparrow_{\text{inst}})$:

$$\begin{aligned}\exists i \in \mathbb{N}.\ \exists \langle \theta, M^{IO}, e_{evt}, w\rangle \in I^{\mathcal{S}}(\mathcal{P} \Uparrow_{\text{inst}}, i).\\ \forall j > i.\ \exists v.\ \langle \theta, M^{IO}, e_{evt}, v\rangle \in I^{\mathcal{S}}(\mathcal{P} \Uparrow_{\text{inst}}, j) \wedge\\ \exists k \geq j.\ \mathsf{ENABLED}_{\text{inst}}(\theta, \mathcal{I}^{\rightarrow}(\mathcal{P} \Uparrow_{\text{inst}}, k))\end{aligned}$$

With $i \in \mathbb{N}$ and $\langle \theta, M^{IO}, e_{evt}, w\rangle \in I^{\mathcal{S}}(\mathcal{P} \Uparrow_{\text{inst}}, i)$, establish the following:

- $\forall m > i.\ \exists u.\ \langle \theta, M^{IO}, e_{evt}, u\rangle \in I^{\mathcal{S}}(\mathcal{P} \Uparrow_{\text{inst}}, m)$:
  Follows directly from the assumption.
- $\exists m > i.\ \forall u.\ \langle \theta, M^{IO}, e_{evt}, u\rangle \notin I^{\mathcal{S}}(\mathcal{P} \Uparrow_{\text{inst}}, m)$:
  Instantiate the assumption with $j = i + 1$.
  Then $\exists k \geq j.\ \mathsf{ENABLED}_{\text{inst}}(\theta, \mathcal{I}^{\rightarrow}(\mathcal{P} \Uparrow_{\text{inst}}, k))$.
  Either $\forall u.\ \langle \theta, M^{IO}, e_{evt}, u\rangle \notin I^{\mathcal{S}}(\mathcal{P} \Uparrow_{\text{inst}}, k+1)$ or $\langle \theta, M^{IO}, e_{evt}, w+1\rangle \in I^{\mathcal{S}}(\mathcal{P} \Uparrow_{\text{inst}}, k+1)$, since $\mathcal{I}^{\rightarrow}(\mathcal{P} \Uparrow_{\text{inst}}, k) \equiv \textsc{SyncCommit}$.
  If $\forall u.\ \langle \theta, M^{IO}, e_{evt}, u\rangle \notin I^{\mathcal{S}}(\mathcal{P} \Uparrow_{\text{inst}}, k+1)$, then $\exists m > i.\ \forall u.\ \langle \theta, M^{IO}, e_{evt}, u\rangle \notin I^{\mathcal{S}}(\mathcal{P} \Uparrow_{\text{inst}}, m)$ follows by taking $m = k + 1$.
  If $\langle \theta, M^{IO}, e_{evt}, w+1\rangle \in I^{\mathcal{S}}(\mathcal{P} \Uparrow_{\text{inst}}, k+1)$, then Lemma 4 with $k+1$, $\langle \theta, M^{IO}, e_{evt}, w+1\rangle \in I^{\mathcal{S}}(\mathcal{P} \Uparrow_{\text{inst}}, k+1)$, and $w + 1 > 0$, establishes $\exists m > k+1.\ \forall u.\ \langle \theta, M^{IO}, e_{evt}, u\rangle \notin I^{\mathcal{S}}(\mathcal{P} \Uparrow_{\text{inst}}, m)$, which implies $\exists m > i.\ \forall u.\ \langle \theta, M^{IO}, e_{evt}, u\rangle \notin I^{\mathcal{S}}(\mathcal{P} \Uparrow_{\text{inst}}, m)$.

This is a contradiction. Therefore, $\mathsf{SYNC\_FAIR}(\mathcal{P} \Uparrow_{\text{inst}})$.