Finitary Fairness in Action Systems

Emil Sekerinski and Tian Zhang

McMaster University, Hamilton, ON, Canada
{emil,zhangt26}@mcmaster.ca

Abstract. In basic action systems, the choice among actions is not restricted. Fairness can be imposed to restrict this nondeterminism. Finitary fairness has been proposed as a further restriction of fairness: it models implementations closer, and allows problems to be solved for which standard fairness is not sufficient. We propose a method for expressing finitary fairness in action systems. We give two general transformations from a system in which some actions are marked as fair, into an equivalent system without fair actions. A theoretical justification is given, and the transformations are illustrated with two examples: alternating bit protocol and distributed consensus. The examples are developed by stepwise refinement in Event-B and are mechanically checked.

Keywords: Finitary Fairness, Modelling, Termination, Event-B, Stepwise Refinement.

1 Introduction

The theory of *action systems* formalizes development of parallel and reactive programs by stepwise refinement [1]. An action system consists of global variables, local variables, an *initialization* statement and a set of *actions* (or atomic guarded commands). Because of existing tool support, we use the Event-B [2] notation for illustration, and we borrow the Event-B term *events* as an alias for actions. A *schedule* is a sequence of event names that can occur in a *computation* (which is going to be made precise). A schedule can be finite or infinite.

Consider the event system in Fig. 1, which is taken from [3]. Both events L and R have no guard and are thus always enabled. In this example all possible schedules are

invariants
  $x \in BOOL$
  $y \in \mathbb{N}$
initialisation
  $x, y := TRUE, 0$
event L
  $x := NOT\ x$
event R
  $y := y + 1$

Fig. 1. A simple event system with two events

Z. Liu, J. Woodcock, and H. Zhu (Eds.): ICTAC 2013, LNCS 8049, pp. 319–336, 2013.
© Springer-Verlag Berlin Heidelberg 2013

infinite. In basic event systems the choice among events is not restricted. Such nondeterministic choice only guarantees *minimal progress*: any enabled event can be taken and an enabled event must be taken only if no other is enabled. With the default assumption of minimal progress, a schedule can contain an infinite sequence of an event, e.g. a schedule that repeatedly executes L after executing L and R twice:

$$LRLRLLLLL\ldots$$

In the design of concurrent systems, *fairness* restricts the nondeterminism leading to minimal progress. It also allows to abstract from scheduling policies in multi-process systems and from processor speeds in multi-processor systems. *Weak fairness* requires that no event can be continuously enabled forever without being taken. This is a useful assumption for multi-process and multi-processor systems: if two continuously enabled events belong to different processes, fairness implies that the scheduler must give each process a chance, without specifying the scheduling policy; if two continuously enabled events are to be executed on different processors, fairness expresses that each processor is working, without quantifying the relative speed.

If both events in the above example are specified to be weakly fair, then fairness of L implies that a schedule cannot contain an infinite sequence of R's, and vice versa. For example, the above schedule would be excluded, but the following schedule, in which the number of consecutive R's continues to increase, is allowed:

$$LRLRRLRRRLRRRRL\ldots$$

A schedule is *weakly k-bounded* if for some natural number k, no fair event is neglected more than k times while being consecutively enabled. *Finitary fairness* of an event system means that all schedules are k-bounded for some $k \in \mathbb{N}$ [3]. The above schedule is not k-bounded for any $k \in \mathbb{N}$; thus the schedule is allowed when L and R are restricted by standard fairness but not when restricted by finitary fairness.

Suppose the events belong to different processes. A *scheduler* is an automaton with event names as the alphabet. For the above schedule to be generated by an automaton, the automaton needs to count the number of R's, so it has an unboundedly large number of states. Conversely, if the schedule is bounded, only a finite number of states are needed. Thus, the bounded schedules are exactly the languages of finite state schedulers. Since any practical scheduler uses a fixed amount of memory, finitary fairness is not only an adequate, but also a more precise abstraction from scheduling policies than standard fairness.

Suppose that the events are executed on different processors; the speeds of the processors may differ and may vary. Finitary fairness implies that the speeds of the processors may not drift apart unboundedly. Alur and Henzinger formalize this claim in terms of timed transition systems [3]. Again, finitary fairness allows a more precise abstraction of multiprocessor systems.

The interleaving model of concurrency represents the concurrent execution of two independent events by a sequential execution in any order. Thus, reasoning about a concurrent system is reduced to reasoning about a nondeterministic sequential system. Since finitary fairness is more restrictive than standard fairness, one can expect more properties to hold under finitary fairness. For example, the event system of Fig. 1 will

eventually reach a state in which $x = TRUE \wedge \neg powerOf2(y)$ holds: if this property would always be false, then L must be scheduled only when $powerOf2(y)$ holds, for increasing values of y, but that is impossible in a bounded schedule.

Finitary fairness allows some problems to be solved for which standard fairness is not sufficient. Furthermore, termination proofs are simpler with finitary fairness compared to standard fairness, as variants need to be over natural numbers only rather than well-founded sets as with standard fairness.

In this paper we propose a method for the stepwise development of action systems with finitary fairness. Finitary fairness is particularly suited for Event-B as in Event-B variants are only over natural numbers. The core contribution of this paper is a transformation of an event system with finitary weak fairness to an equivalent one without fairness. A similar transformation was proposed in [3], but that transformation does not result in an equivalent system, only in one in which all computations terminate if and only if all finitary weakly fair computations of the original system do (which was the intention). The transformed system does not have any restriction on the counters, so it may enter a state in which two counters reach the upper limit at the same time and then forced to terminate prematurely. Consequently, it does not have the same computations as the original system. The theoretical finitary restriction, as proposed in [3], can be applied to both weak fairness and *strong fairness*, which differs from weak fairness by requiring that an event must be taken if it is enabled repeatedly, but not necessarily continuously. In this work, we restrict our transformation to weak fairness as in [3], as for the example at hand, distributed consensus, weak fairness is sufficient.

Section 2 summarizes related work. Section 3 formally defines transition systems with fair events, computations, and finitary fairness. Section 4 presents two methods for transforming an event system with fair and regular events to one that has only regular events but produces the same computations. In Sect. 5 one of the two transformations is illustrated in the stepwise refinement of the alternating bit protocol, an example that has been studied repeatedly in literature; fairness is needed as the assumption that a message in transmission will not always be lost and has a fair chance of reaching the destination. In Sect. 6 the other transformation is illustrated in the stepwise refinement of a distributed consensus algorithm, an example in which finitary fairness can guarantee termination, but standard fairness cannot. The final section gives an outlook.[1]

2 Related Work

Programming theories with fairness are well worked out, e.g. [5,6,7]. Extensions of action systems to fairness have been proposed [8,9,10]. In [9] refinement rules that preserve temporal (leads-to) and fixpoint (termination) properties are studied for fair transitions systems (action systems). Here we restrict ourselves to terminating action systems but consider local variables, allowing a more general notion of refinement.

The approach of [10] is to augment action systems by explicitly specifying and prohibiting unfair non-terminating computations, rather than assuming a fair choice among

[1] The models in this paper are developed in Event-B using the Rodin platform [4], an Eclipse-based IDE for Event-B. All proof obligations have been successfully proved. The Rodin project files are available at `http://www.cas.mcmaster.ca/~zhangt26/ICTAC/`

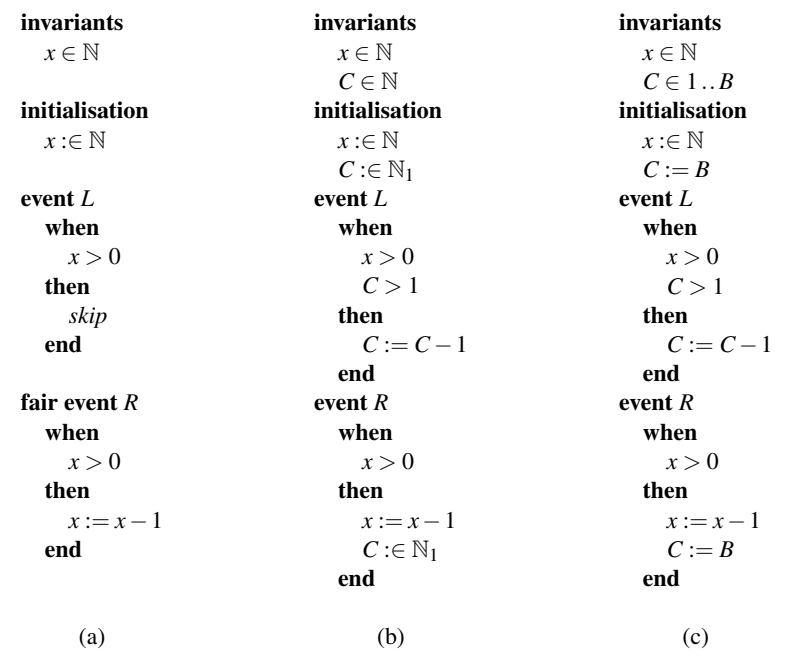

Fig. 2. (a) Event system with fair event R. (b) The counter C is used to ensure standard fairness of R. (c) The counter C is used to ensure finitary fairness of R.

actions, and to study refinement of such augmented action systems; this allows a wider range of fairness constraints to be expressed compared to the (weak) fairness considered here, although in a different style.

The proof rule for the termination of an event system is more involved in the presence of fairness: events either must decrease the variant or keep fair events which decrease the variant enabled, as by fairness these will eventually be taken. The proof rule requires that an invariant is specified for each event, e.g. as in [8] for the refinement of action systems, rather than one invariant for the whole system as in Event-B. This would require the proof rules of Event-B to be significantly expanded.

The alternative that we follow is to transform an event system by replacing fair events with regular events and introducing an explicit scheduler [11,1]. The standard proof rules of Event-B can then be applied. Figure 2 illustrates this. The event system of (a) will eventually terminate as x is set initially to some natural number, and fair event R always decreases x. In both transformed event systems (b) and (c) fairness is achieved by introducing a (down-) counter C that is decreased each time the (regular) event L is taken. This eventually forces R to be taken as L becomes disabled when C reaches 1. When R is taken, C is reset. In (b) the (down-) counter C does not have an upper bound, but still event R will eventually be taken; this ensures standard fairness. In (c), C is at most B, so $B-1$ is the upper bound of how many times event R can be consecutively ignored before it must be taken, hence the schedules are $(B-1)$-bounded and the model ensures finitary fairness.

A further reason for preferring finitary fairness is that it can simplify proofs of termination. For a set of events to terminate, there must exist a variant, which is a function from the states to a well-founded domain, and all events have to decrease the variant. For proving the termination of the event system in Fig. 2 (c), the following variant with natural numbers as the well-founded domain is sufficient:

$$\textbf{variant}\quad x*B+C$$

Event L decreases the variant by decreasing C. Event R decreases the variant by decreasing x ($C \in 1..B$ so the variant is still decreased even when C is reset to B). A similar variant cannot be given for the event system in (b): natural numbers as the well-founded domain are not sufficient with standard fairness.

The finitary restriction can be used for modelling *unknown delays* of timed systems. In the problem of distributed consensus, processes have to agree on a common output value, but processes with different preferences may try to set the output at the same time. Besides, each process may fail and not deliver a value. This can be solved using finitary fairness, as shown in [3], but cannot be solved using standard fairness only [12].

3 Fair Event Systems

When considering finitary fairness, we are interested only in k-bounded computations. Following definition of fair event systems generalizes that of transitions systems in [3] by indexing the transitions with the events and by allowing only some events to be fair.

Definition 1. *A* fair event system *P is a structure (Q,I,F,E,T) where*

- *Q is the set of states,*
- *I is the set of initial states, $I \subseteq Q$,*
- *F is the number of fair events,*
- *E is the set of all events, with cardinality $N \geq F$ (we assume that e_i is a fair event for $i \in 1..F$ and a regular event for $i \in F+1..N$),*
- *T is the set of transitions, relations over $Q \times Q$ indexed by E.*

We write $T(e)$ for the transition relation of event e. A computation *comp* of P is a finite or infinite maximal sequence of states and events alternating, written

$$comp = \sigma_0 \xrightarrow{\tau_0} \sigma_1 \xrightarrow{\tau_1} \sigma_2 \xrightarrow{\tau_2} \cdots$$

such that $\sigma_0 \in I$ and $\forall i \cdot i \in \mathbb{N} \Rightarrow \tau_i \in E \wedge \sigma_i \mapsto \sigma_{i+1} \in T(\tau_i)$. That is, states σ_i and σ_{i+1} must be in relation $T(\tau_i)$. A computation is a finite sequence, or is *terminating*, if it ends with a state σ_n, for some $n \in \mathbb{N}$, that is not in the domain of any transition relation, i.e. $\forall e \cdot e \in E \Rightarrow \sigma_n \notin dom(T(e))$. Otherwise it is an infinite sequence, or is *nonterminating*. The schedule of a computation *comp* is the projection of the sequence *comp* to the events; the *trace* of *comp* is the projection of *comp* to the states:

$$schedule(comp) = \tau_0\tau_1\tau_2\ldots \qquad trace(comp) = \sigma_0\sigma_1\sigma_2\ldots$$

We write $schedule_i(comp)$ for τ_i, the i-th event of computation $comp$, and $trace_i(comp)$ for σ_i, the i-th state of computation $comp$. The *guard* of an event is the domain of its relation, $grd(e) = dom(T(e))$; an event is *enabled* in a state if the state is in its guard, otherwise *disabled*. A computation $comp$ is *bounded* if it is finite or if for some $k \in \mathbb{N}$, any fair event e_f, for some $f \in 1..F$, cannot be enabled for more than k consecutive states without being taken, formally:

$$\forall i,f \cdot i \in \mathbb{N} \land f \in 1..F \Rightarrow \\ \quad \exists j \cdot j \in i..i+k \land (schedule_j(comp) = e_f \lor trace_j(comp) \notin grd(e_f))$$

An Event-B model defines the set of states through the variables and invariants, the transition relations through guards and generalized substitutions, and the initial states through the initialization event. Thus, fair event systems are abstract representations of Event-B models, in which we additionally allow some events to be specified as fair.

4 The Finitary Weakly Fair Transformation

Let $P = (Q,I,F,E,T)$ be a fair event system. Now we give the definition of the *finitary weakly fair transformation*, the application of which to P written as $fwf(P)$. It expresses finitary fairness by introducing (down-) counter variables $C_1,\ldots,C_F$, one for each fair event. The counter C_i, for $i \in 1..F$, of event e_i indicates that e_i must be taken or disabled at least once in the next C_i transitions. Once the counter of an event reaches 1, that event must be tested: if it is enabled, it must be taken, otherwise it may be skipped.

Care is needed to avoid that several counters reach 1 simultaneously, as the corresponding events cannot be taken in one transition. A naive approach would be to initialize them to distinct values between 1 and B and keep them distinct by decreasing all by 1 simultaneously and cyclically reset them to B: that would enforce round-robin scheduling, but that is too restrictive as we do not want to preclude any fair schedule. To obtain an appropriate translation, in $fwf(P)$ we add a permutation p of $1..F$, on the basis of $fin(P)$ in [3]. The permutation p satisfies $\forall j \cdot j \in 1..F \Rightarrow C_{p(j)} \geq j$ in every state, to guarantee that only one counter can be 1. On every transition, the guards of all fair events must be tested: if a fair event is enabled but not taken, its counter must be decreased, otherwise its counter is reset to B. The counters are all initialized to have the value B; they do not have to be distinct. Using p keeps the system in safe states, while the set of possible schedules remains the same.

Definition 2. *For fair event system P, the finitary weak fair transformation* $fwf(P) = (Q',I',0,E,T')$ *is given by:*

- $Q' = Q \times [F,+\infty) \times [1,B]^F$
- *For every event* $e_i \in E$, $(\sigma,B,C_1,\ldots,C_F) \mapsto (\sigma',B',C'_1,\ldots,C'_F) \in T'(e_i)$ *if:*
  1. $B = B' \land \sigma \mapsto \sigma' \in T(e_i) \land (i \in 1..F \Rightarrow C'_i = B)$
  2. $\forall j \cdot j \in 1..F \setminus \{i\} \Rightarrow$
     $((\sigma \in grd(e_j) \land C'_j \geq 1 \land C'_j = C_j - 1) \lor (\sigma \notin grd(e_j) \land C'_j = B))$
  3. *A permutation* p *of* $1..F$, *exists, such that* $\forall j \cdot j \in 1..F \Rightarrow C'_{p(j)} \geq j$
- I' *is such that* $(\sigma,B,C_1,\ldots,C_F) \in I'$ *if*

1. $\sigma \in I \wedge B \geq F$
2. $\forall j \cdot j \in 1..F \Rightarrow C_j = B$

All counters of the finitary fair transformation are between 1 and B, and the permutation p always exists. That is, for all computations *comp* of $fwf(P)$, for all natural numbers i with $0 \leq i < |trace(comp)|$, and writing $\rightarrowtail\!\!\!\!\!\twoheadrightarrow$ for the type of bijective functions:

$$\begin{array}{l} trace_i(comp) = (\sigma, B, C_1, \ldots, C_F) \Rightarrow \\ \quad \exists p \cdot p \in 1..F \rightarrowtail\!\!\!\!\!\twoheadrightarrow 1..F \wedge (\forall j \cdot j \in 1..F \Rightarrow C_j \in 1..B \wedge C_{p(j)} \geq j) \end{array} \tag{1}$$

This property follows by induction over i: with $fwf(P) = (Q', I', 0, E, T')$ the initial states in I' satisfy (1) and transitions in T' preserve (1).

Compared to $fin(P)$ in [3], the resulting system prevents premature termination, because it guarantees that as long as one event is enabled in some state σ_i ($i \in \mathbb{N}$) in some computation *comp* of P, then at least one event is enabled in the corresponding state σ'_i in *comp* of $fwf(P)$. If all fair events are disabled, then the additional guards of regular events are always satisfied, i.e. then the additional guards will not prevent any enabled regular event from being taken.

The introductory example results from this transformation. In practice, this transformation has the drawback that a term for stating the existence of a permutation will increase exponentially with the number of fair events. In the following alternative transformation, $dist(P)$, all counters are kept distinct. A term stating the distinctness of F counters would have $(F * (F+1)/2)$ clauses, which with large number of fair events results in significantly more compact descriptions.

Definition 3. *For fair event system P, the finitary weak fair transformation* $dist(P) = (Q', I', 0, E, T')$ *is given by:*

- $Q' = Q \times [F, +\infty) \times [1, B]^F$
- *For every event* $e_i \in E$, $(\sigma, B, C_1, \ldots, C_F) \mapsto (\sigma', B', C'_1, \ldots, C'_F) \in T'(e_i)$ *if:*
  1. $B = B' \wedge \sigma \mapsto \sigma' \in T(e_i) \wedge (i \in 1..F \Rightarrow C'_i \in 1..B)$
  2. $\forall j \cdot j \in 1..F \backslash \{i\} \Rightarrow C'_j \in 1..B \wedge (\sigma \in grd(e_j) \Rightarrow C'_j = C_j - 1)$
  3. $distinct(C'_1, \ldots, C'_F)$
- I' *is such that* $(\sigma, B, C_1, \ldots, C_F) \in I'$ *if*
  1. $\sigma \in I \wedge B \geq F$
  2. $\forall j \cdot j \in 1..F \Rightarrow C_j \in 1..B$
  3. $distinct(C_1, \ldots, C_F)$

All counters of $dist(P)$ are between 1 and B and are distinct, i.e. for all computations *comp* of $fwf(P)$ and for all natural numbers i with $1 \leq i < |trace(comp)|$:

$$\begin{array}{l} trace_i(comp) = (\sigma, B, C_1, \ldots, C_F) \Rightarrow \\ \quad (\forall j \cdot j \in 1..F \Rightarrow C_j \in 1..B) \wedge distinct(C_1, \ldots, C_F) \end{array} \tag{2}$$

This property follows by induction over i: with $dist(P) = (Q', I', 0, E, T')$ the initial states in I' satisfy (2) and transitions in T' preserve (2).

Theorem 1. *For any fair event system P, the schedules of* $fwf(P)$ *and of* $dist(P)$ *are exactly the finitary weak fair schedules of P.*

Proof. Let P be (Q, I, F, E, T). We give the proof only for $fwf(P)$; the one for $dist(P)$ has the same structure. The proof proceeds by mutual inclusion. For showing that the schedules of $fwf(P)$ are bounded schedules of P, we note that the schedules of $fwf(P)$ are also schedules of P, so it remains to be shown that they are bounded. We reformulate the definition of a bounded computation. A computation *comp* is bounded if it is finite or for some $k \in \mathbb{N}$, for all $e \in F$, for all $i \in \mathbb{N}$:

$$(\forall j \cdot j \in i \mathinner{..} i+k \Rightarrow trace_j(comp) \in grd(e)) \Rightarrow \exists j \cdot j \in i \mathinner{..} i+k \wedge schedule_j(comp) = e$$

Let *comp* be a computation of $fwf(P)$. It is obvious that if $schedule(comp)$ is finite, it is bounded with $k = B-1$ by definition. When it is infinite, let e_f be a fair event ($f \in 1 \mathinner{..} F$), let i be a natural number, and assume that $\forall j \cdot j \in i \mathinner{..} i+k \Rightarrow trace_j(comp) \in grd(e_f)$. We have to show that $schedule_j(comp) = e_f$ for some $j \in i \mathinner{..} i+k$. We prove this by contradiction. If such an index j does not exist, then since e_f is consecutively enabled, according to the properties of transitions in $fwf(P)$, in every step C_f is decremented by 1. After B steps, C_f is decreased by B. In state $trace_i$, $C_f \in 1 \mathinner{..} B$, so in state $trace_{i+k+1}(comp)$, $C_f \in 1-B \mathinner{..} 0$, which contradicts the constraint $C_f \in 1 \mathinner{..} B$. Hence after at most B transitions, event e_f must have been taken.

Now let *comp* be a bounded computation of P. We have to show that a computation $comp'$ of $fwf(P)$ exists such that $schedule(comp')$ equals $schedule(comp)$. During initialization, if B is picked such that $B \geq k+F$, then all the counters are always greater or equal to F in *comp* (since *comp* is k-bounded, then every counter in $fwf(P)$ is reset at least once in every $k+1$ steps, either due to being disabled or executed), so such a permutation p always exists throughout the schedule (simply *id* on $1 \mathinner{..} F$), and no event will be disabled by the additional guards; thus a computation $comp'$ which yields the same schedule does exist in $fwf(P)$. This completes the proof.

5 The Alternating Bit Protocol

The alternating bit protocol (ABP) [13], a protocol for reliable communication over unreliable channels, has repeatedly been formalized. Our treatment is inspired by that of [5,14]. Channels are modelled as simple variables, as in [15,16], rather than as sequences. The first refinement step is similar to the file transfer example of [2]. The refinement process is illustrated in Fig. 3. It uses the $dist(P)$ transformation.

Specification. In its most abstract form, a transmission copies sequence a to sequence z. We let *SIZE* be a positive natural number and *DATA* a set.

MACHINE ABP0
SEES Context
VARIABLES
  z target file
  e mark of termination
INVARIANTS
  inv1 : $z \in 1 \mathinner{..} SIZE \rightarrow DATA$
  inv2 : $e \in 0 \mathinner{..} 1$
  inv3 : $e = 0 \Rightarrow z = a$
EVENTS
Initialisation
  begin
    act1 : $z := \varnothing$
    act2 : $e := 1$
  end
Event *TransferAll* $\widehat{=}$
Status convergent
  when
    grd1 : $e = 1$
  then
    act1 : $z := a$
    act2 : $e := 0$
  end
VARIANT
  e
END

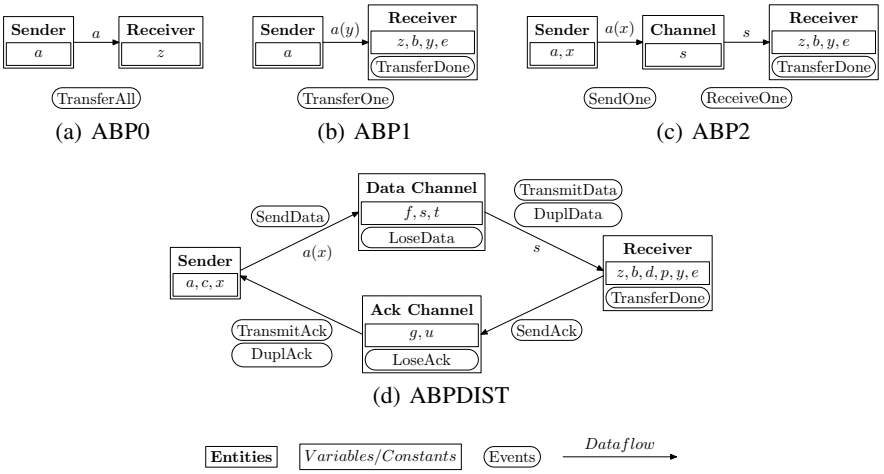

Fig. 3. Refinement process of ABP

Copying Data Items Successively. In the first refinement step, the data items are copied one by one by a new *convergent* event.

MACHINE ABP1
REFINES ABP0
SEES Context
VARIABLES
`z` target file
`b` received part
`y` index
`e` mark of termination
INVARIANTS
`inv1`: $b \in 1..SIZE \rightarrow DATA$
`inv2`: $y \in 1..SIZE + 1$
`inv3`: $b = 1..y-1 \lhd a$
EVENTS
Initialisation
begin
`act1`: $z := \varnothing$
`act2`: $b := \varnothing$
`act3`: $y := 1$
`act4`: $e := 1$
end

Event *TransferOne* $\widehat{=}$
Status convergent
when
`grd1`: $y \leq SIZE$
then
`act1`: $b := b \cup \{y \mapsto a(y)\}$
`act2`: $y := y + 1$
end
Event *TransferDone* $\widehat{=}$
refines *TransferAll*
when
`grd1`: $y = SIZE + 1$
`grd2`: $e = 1$
then
`act1`: $z := b$
`act2`: $e := 0$
end
VARIANT
`SIZE` $+ 1 -$ `y`
END

Introducing Data Channel. A variable s is introduced into which the sender writes the next data item and the receiver reads from. Sender and receiver maintain their own count of the number of items sent and received in the variables x and y. Sending and reading proceeds in a ping-pong fashion, controlled by $y - x$.

MACHINE ABP2
REFINES ABP1
SEES Context
VARIABLES
`z` target file
`b` received part
`s` the unit of info sent
`x` sender index
`y` receiver index
`e` mark of termination
INVARIANTS
`inv1`: $s \in DATA$
`inv2`: $x \in 1..SIZE + 1$
`inv3`: $x = y \vee x = y + 1$
`inv4`: $x = y + 1 \Rightarrow s = a(y)$
EVENTS
Initialisation
begin

act1 : $z := \varnothing$
act2 : $b := \varnothing$
act3 : $s :\in DATA$
act4 : $x, y := 1, 1$
act5 : $e := 1$
end
Event *SendOne* $\widehat{=}$
Status convergent
when
grd1 : $x = y \wedge x \leq SIZE$
then
act1 : $s := a(x)$
act2 : $x := x + 1$
end
Event *ReceiveOne* $\widehat{=}$
refines *TransferOne*
when
grd1 : $x = y + 1$
then
act1 : $b := b \cup \{y \mapsto s\}$
act2 : $y := y + 1$
end
Event *ReceiverDone* $\widehat{=}$
refines *TransferDone*
when
grd1 : $y = SIZE + 1$
grd2 : $e = 1$
then
act1 : $z := b$
act2 : $e := 0$
end
VARIANT
$y + 1 - x$
END

Introducing Faulty Channels. The sender places the data to be transmitted in the variable s and sets the flag for transmitting, t to 1. The sender also flips its *alternating bit*, c. Event *TransmitData* represents successful transmission in which c and s are copied to f and p, then disables itself by resetting t to 0; the event *DuplicateData* copies the data but does not disable itself; the event *LoseData* does not even copy the data. Successful data transmission is only possible if *TransmitData* is fair. On receiving a new value of f, the receiver appends p to the data received so far and flips its alternating bit g. The acknowledgement channel works analogously to the data channel. Using a hypothetical extension of Event-B with fair events, this is expressed as:

MACHINE ABPFAIR
REFINES ABP2
SEES Context
VARIABLES
z target file
b received part
c sender private bit
d receiver private bit
f data channel signal bit
g ack channel signal bit
p data of data channel
s data of sender
t data transmitting signal
u ack transmitting signal
x sender index
y receiver index
e mark of task finished
INVARIANTS
inv1 : $c \in 0..1$
inv2 : $d \in 0..1$
inv3 : $f \in 0..1$
inv4 : $g \in 0..1$
inv5 : $p \in DATA$
inv6 : $t \in 0..1$
inv7 : $u \in 0..1$
inv8 : $c = g \vee c \neq f \vee d = f \vee d = g$
four states: ready to send data; data sent and to be transmitted; ready to receive data; data received and ack to be transmitted;
inv9 : $c = g \Rightarrow c \neq d \wedge c = f \wedge x = y$
ready to send data
inv10 : $c \neq f \Rightarrow c = d \wedge c \neq g \wedge x = y + 1 \wedge s = a(y)$
data sent and to be transmitted
inv11 : $d = f \Rightarrow c = d \wedge c \neq g \wedge x = y + 1 \wedge s = a(y) \wedge p = s$
ready to receive data
inv12 : $d = g \Rightarrow c \neq d \wedge c = f \wedge x = y$
data received and Ack to be transmitted

EVENTS
Initialisation
begin
act1 : $z := \varnothing$
act2 : $b := \varnothing$
act3 : $c := 1$
act4 : $d := 0$
act5 : $f := 1$
act6 : $g := 1$
act7 : $p :\in DATA$
act8 : $s :\in DATA$
act9 : $t, u, x, y, e := 0, 0, 1, 1, 1$
end
Event *SendData* $\widehat{=}$
refines *SendOne*
when
grd1 : $c = g \wedge x \leq SIZE$
then
act1 : $c, s, t, x, u := 1 - c, a(x), 1, x + 1, 0$
end
FAIR Event *TransmitData* $\widehat{=}$
Status convergent
when
grd1 : $t = 1$
then
act1 : $f, p, t := c, s, 0$
end
Event *DuplData* $\widehat{=}$
Status convergent
when
grd1 : $t = 1$
then
act1 : $f, p := c, s$
end
Event *LoseData* $\widehat{=}$
Status convergent
when
grd1 : $t = 1$
then
skip
end

Event *SendAck* $\widehat{=}$
refines *ReceiveOne*
  when
    grd1 : $d = f$
  then
    act1 : $b,d,u,y,t := b \cup \{y \mapsto p\}, 1-d, 1, y+1, 0$
  end
FAIR Event *TransmitAck* $\widehat{=}$
Status convergent
  when
    grd1 : $u = 1$
  then
    act1 : $g,u := 1-d, 0$
  end
Event *DuplAck* $\widehat{=}$
Status convergent
  when
    grd1 : $u = 1$
  then
    act1 : $g := 1-d$
  end
Event *LoseAck* $\widehat{=}$
Status convergent
  when
    grd1 : $u = 1$
  then
    skip
  end
Event *AllReceived* $\widehat{=}$
refines *ReceiverDone*
  when
    grd1 : $y = SIZE + 1$
    grd2 : $e = 1$
  then
    act1 : $z := b$
    act2 : $e := 0$
  end
END

Now we apply *dist* to *ABPFAIR*. For expressing the result in Event-B we use following scheme. Suppose L is a regular event and $R1, R2$ are fair events. Three variables, B, $C1$ and $C2$, are introduced and $F = 2$ is the number of fair events:

event L
  when
    g
  then
    S
  end
fair event $R1$
  when
    $h1$
  then
    $T1$
  end
fair event $R2$
  when
    $h2$
  then
    $T2$
  end

invariant
  $C1 \in 1..B$
  $C2 \in 1..B$
  $C1 \neq C2$
initialisation
  $B, C1, C2 : |B' \geq F \wedge C1' \in 1..B' \wedge C2' \in 1..B' \wedge C1' \neq C2'$
event L
  when
    g
    $min(\{C1, C2\}) > 1 \vee (C1 = 1 \wedge \neg h1) \vee (C2 = 1 \wedge \neg h2)$
  then
    S
    $C1, C2 : | C1' \in 1..B \wedge C2' \in 1..B \wedge C1' \neq C2' \wedge$
      $(h1 \Rightarrow C1' = C1 - 1) \wedge (h2 \Rightarrow C2' = C2 - 1)$
  end
event $R1$
  when
    $h1$
    $h2 \Rightarrow C2 - 1 \geq 1$
  then
    $T1$
    $C1, C2 : | C1' \in 1..B \wedge C2' \in 1..B \wedge C1' \neq C2' \wedge (h2 \Rightarrow C2' = C2 - 1)$
  end
...

It is easy to see that this scheme satisfies the conditions of *dist*. If no fair event is enabled, the additional guard of regular events is satisfied, formally:

$$(\neg h1 \wedge \neg h2) \Rightarrow (min(\{C1, C2\}) > 1 \vee (C1 = 1 \wedge \neg h1) \vee (C2 = 1 \wedge \neg h2))$$

This prevents the premature termination of $fin(P)$ in [3]. In the application of this scheme to *ABPFAIR*, we additionally introduce a counter *step* as a "ghost variable" for proving termination. As a note, the development provides a lower bound of the number of steps for termination (*inv*20 below).

MACHINE ABPDIST
REFINES ABP2
SEES Context
VARIABLES
  z target file
  b received part
  c sender private bit
  d receiver private bit
  f data channel signal bit
  g ack channel signal bit
  p data of data channel
  s data of sender

`t` data transmitting signal
`u` ack transmitting signal
`x` sender index
`y` receiver index
`e` mark of task finished
`B` bound
`C1` down-counter of fair event *TransmitData*
`C2` down-counter of fair event *TransmitAck*
`step` step counter

INVARIANTS

`inv1`: $c \in 0..1$
`inv2`: $d \in 0..1$
`inv3`: $f \in 0..1$
`inv4`: $g \in 0..1$
`inv5`: $p \in DATA$
`inv6`: $t \in 0..1$
`inv7`: $u \in 0..1$
`inv8`: $c = g \vee c \neq f \vee d = f \vee d = g$
four states: ready to send data; data sent and to be transmitted; ready to receive data; data received and ack to be transmitted;
`inv9`: $c = g \Rightarrow c \neq d \wedge c = f \wedge x = y$
ready to send data
`inv10`: $c \neq f \Rightarrow c = d \wedge c \neq g \wedge x = y+1 \wedge s = a(y)$
data sent and to be transmitted
`inv11`: $d = f \Rightarrow c = d \wedge c \neq g \wedge x = y+1 \wedge s = a(y) \wedge p = s$
ready to receive data
`inv12`: $d = g \Rightarrow c \neq d \wedge c = f \wedge x = y$
data received and Ack to be transmitted
`inv13`: $B \geq F$
`inv14`: $C1 \in 1..B$
`inv15`: $C2 \in 1..B$
`inv16`: $C1 \neq C2$
always distinct
`inv17`: $step \in \mathbb{N}$
`inv18`: $step \leq (x+y-2)*(B+1) - t*C1 - u*C2 + 1 - e$
the strict upper bound for proving that the variant is non-negative
`inv19`: $step \geq 2*(x+y) - (d+g-1)*(d+g-1) - (c-f)*(c-f) - e - 3$
the strict lower bound of *step*, for proving *inv*20
`inv20`: $e = 0 \Rightarrow step \geq 4*SIZE$
the strict lower bound of *step* when the system terminates (exactly $4*SIZE$ when there is no duplication, no loss, and the last event is *AllReceived*, right after event *SendAck*)

EVENTS

Initialisation
begin
`act1`: $z := \varnothing$
`act2`: $b := \varnothing$
`act3`: $c := 1$
`act4`: $d := 0$
`act5`: $f := 1$
`act6`: $g := 1$
`act7`: $p :\in DATA$
`act8`: $s :\in DATA$
`act9`: $t,u,x,y,e := 0,0,1,1,1$
`act10`: $B,C1,C2 : |B' \geq F \wedge C1' \in 1..B' \wedge C2' \in 1..B' \wedge C1' \neq C2'$
`act11`: $step := 0$
end

Event *SendData* $\widehat{=}$
refines *SendOne*
when
`grd1`: $c = g \wedge x \leq SIZE$
`grd2`: $min(\{C1,C2\}) > 1 \vee (C1 = 1 \wedge t = 0) \vee (C2 = 1 \wedge u = 0)$
then
`act1`: $c,s,t,x,u := 1-c, a(x), 1, x+1, 0$
`act2`: $C1,C2 : |C1' \in 1..B \wedge C2' \in 1..B \wedge C1' \neq C2' \wedge (t = 1 \Rightarrow C1' = C1-1) \wedge (u = 1 \Rightarrow C2' = C2-1)$
`act3`: $step := step + 1$
end

Event *TransmitData* $\widehat{=}$
Status convergent
when
`grd1`: $t = 1$
`grd2`: $u = 1 \Rightarrow C2 - 1 \geq 1$
then
`act1`: $f,p,t := c,s,0$
`act2`: $C1,C2 : |C1' \in 1..B \wedge C2' \in 1..B \wedge C1' \neq C2' \wedge (u = 1 \Rightarrow C2' = C2-1)$
`act3`: $step := step + 1$
end

Event *DuplData* $\widehat{=}$
Status convergent
when
`grd1`: $t = 1$
`grd2`: $min(\{C1,C2\}) > 1 \vee (C1 = 1 \wedge t = 0) \vee (C2 = 1 \wedge u = 0)$
then
`act1`: $f,p := c,s$
`act2`: $C1,C2 : |C1' \in 1..B \wedge C2' \in 1..B \wedge C1' \neq C2' \wedge (t = 1 \Rightarrow C1' = C1-1) \wedge (u = 1 \Rightarrow C2' = C2-1)$
`act3`: $step := step + 1$
end

Event *LoseData* $\widehat{=}$
Status convergent
when
`grd1`: $t = 1$
`grd2`: $min(\{C1,C2\}) > 1 \vee (C1 = 1 \wedge t = 0) \vee (C2 = 1 \wedge u = 0)$
then
`act1`: $C1,C2 : |C1' \in 1..B \wedge C2' \in 1..B \wedge C1' \neq C2' \wedge (t = 1 \Rightarrow C1' = C1-1) \wedge (u = 1 \Rightarrow C2' = C2-1)$
`act2`: $step := step + 1$
end

Event *SendAck* $\widehat{=}$
refines *ReceiveOne*
when
`grd1`: $d = f$
`grd2`: $min(\{C1,C2\}) > 1 \vee (C1 = 1 \wedge t = 0) \vee (C2 = 1 \wedge u = 0)$
then
`act1`: $b,d,u,y,t := b \cup \{y \mapsto p\}, 1-d, 1, y+1, 0$
`act2`: $C1,C2 : |C1' \in 1..B \wedge C2' \in 1..B \wedge C1' \neq C2' \wedge (t = 1 \Rightarrow C1' = C1-1) \wedge (u = 1 \Rightarrow C2' = C2-1)$
`act3`: $step := step + 1$
end

Event *TransmitAck* $\widehat{=}$
Status convergent
when
`grd1`: $u = 1$
`grd2`: $t = 1 \Rightarrow C1 - 1 \geq 1$
then
`act1`: $g,u := 1-d, 0$
`act2`: $C1,C2 : |C1' \in 1..B \wedge C2' \in 1..B \wedge C1' \neq C2' \wedge (t = 1 \Rightarrow C1' = C1-1)$
`act3`: $step := step + 1$
end

Event *DuplAck* $\widehat{=}$
Status convergent
when
`grd1`: $u = 1$

grd2 : $min(\{C1, C2\}) > 1 \lor$
$(C1 = 1 \land t = 0) \lor$
$(C2 = 1 \land u = 0)$
then
act1 : $g := 1 - d$
act2 : $C1, C2 : |C1' \in 1..B \land C2' \in 1..B \land$
$C1' \neq C2' \land (t = 1 \Rightarrow C1' = C1 - 1) \land$
$(u = 1 \Rightarrow C2' = C2 - 1)$
act3 : $step := step + 1$
end
Event *LoseAck* $\widehat{=}$
Status convergent
when
grd1 : $u = 1$
grd2 : $min(\{C1, C2\}) > 1 \lor$
$(C1 = 1 \land t = 0) \lor$
$(C2 = 1 \land u = 0)$
then
act1 : $C1, C2 : |C1' \in 1..B \land C2' \in 1..B \land$
$C1' \neq C2' \land (t = 1 \Rightarrow C1' = C1 - 1) \land$
$(u = 1 \Rightarrow C2' = C2 - 1)$
act2 : $step := step + 1$
end
Event *AllReceived* $\widehat{=}$
refines *ReceiverDone*
when
grd1 : $y = SIZE + 1$
grd2 : $e = 1$
grd3 : $min(\{C1, C2\}) > 1 \lor$
$(C1 = 1 \land t = 0) \lor$
$(C2 = 1 \land u = 0)$
then
act1 : $z := b$
act2 : $e := 0$
act3 : $C1, C2 : |C1' \in 1..B \land C2' \in 1..B \land$
$C1' \neq C2' \land (t = 1 \Rightarrow C1' = C1 - 1) \land$
$(u = 1 \Rightarrow C2' = C2 - 1)$
act4 : $step := step + 1$
end
VARIANT
$2 * \mathtt{SIZE} * (\mathtt{B} + 1) + 1 - \mathtt{step}$
END

6 Distributed Consensus

Given a group of initial values (in the simplest case, 0 and 1), distributed consensus is to let a group of processes decide by themselves and finally agree on a value. In this section, we model the algorithm in [3] and prove its termination in Event-B. It is an example in case, because its termination can be guaranteed with finitary fairness but not with standard fairness [3]. The refinement process is illustrated in Fig. 5.

The original algorithm is in Fig. 4. The algorithm proceeds in rounds, using a two-dimensional bit array $x[*, 2]$ and an infinite array $y[*]$ over values $\perp$, 0, or 1. When the processes agree on a value, the decision, it is written to the shared bit *out*, the initial value of which is $\perp$. Each process P_i has a local register v_i of its current preference and a local register r_i of its current round number. The ith process has an initial input in_i. If in the rth round, all processes have the same preference v, then they decide on the value v in round r. Only when two processes with different preferences both find $y[r] = \perp$ (line 3), and one of them proceeds and chooses its preference for the next round (line 7) before the other one finishes the assignment to $y[r]$, there is a conflict, and the processes continue trying to resolve it in the next round.

In line 5, the empty loop runs for r_i times before copying its preference of next round from $y[i]$, trying to give other processes time to write their preferences into $y[i]$ first.

Shared registers : initially $out = \perp, y[1..] = \perp, x[1.., 0..1] = 0$;
Local registers : initially $r_i = 1, v_i = in_i$;
1. while $out = \perp$ do
2. $x[r_i, v_i] := 1$;
3. if $y[r_i] = \perp$ then $y[r_i] := v_i$ fi;
4. if $x[r_i, \neg v_i] = 0$ then $out := v_i$
5. else for $j = 1$ to r_i do skip od;
6. $v_i := y[r_i]$;
7. $r_i := r_i + 1$
8. fi
9. od;
10. $decide(out)$.

Fig. 4. The original distributed consensus algorithm in [3]

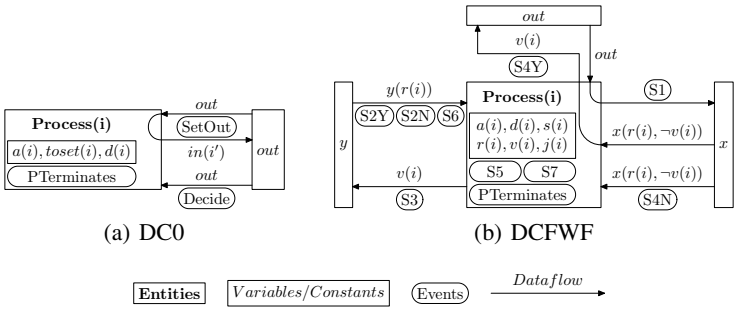

Fig. 5. Refinement process of distributed consensus (*read* operations in guards included)

Standard fairness is not enough to guarantee consensus in any round since theoretically the scheduler might pick this process and continue executing it until the end of the loop, which leaves the conflict unresolved. However, with finitary fairness, when r_i reaches a certain value, the fairness restriction would ensure that every other process gets its chance to finish writing to $y[i]$, thus guarantees that a consensus will be reached in that round.

We translate the algorithm into an event system by breaking down every atomic operation into one event, and adding a state variable s_i for the ith process to record which atomic operation it is to execute next.

Specification. In its most abstract form, *read*, *write* and "*read&write*" on shared registers are all atomic. We let N be the number of processes. The activenesses of processes are represented by the variable a, which also serves as the guard of process actions. The failures of processes are modelled by event *PTerminates*, to show that the model is fault-tolerant (as long as some processes survive, they will finally reach a consensus).

MACHINE DC0
SEES Context
VARIABLES
`a` activeness of processes
`out` the output
`toset` signs of whether a process has not assigned a value to *out* yet
`d` decisions of processes
INVARIANTS
`inv1`: $a \in 1..N \to 0..1$
1 means active and 0 means inactive
`inv2`: $out \in -1..1$
all -1s stand for undefined in this model
`inv3`: $toset \in 1..N \to 0..1$
1 means has not assigned to *out* yet and 0 means has
`inv4`: $d \in 1..N \to -1..1$
`inv5`: $\forall n \cdot n \in 1..N \Rightarrow d(n) \in \{-1, out\}$
agreement: all decisions made by processes must be the same (as *out*)
`inv6`: $out = -1 \vee (\exists n \cdot n \in 1..N \wedge out = in(n))$
validity: the decision must equal to one of the inputs
EVENTS
Initialisation
begin
`act1`: $a := 1..N \times \{1\}$
all active
`act2`: $out := -1$
out undefined
`act3`: $toset := 1..N \times \{1\}$
no process has assigned a value to *out* yet
`act4`: $d := 1..N \times \{-1\}$
decisions all undefined
end
Event *SetOut* $\widehat{=}$
Status convergent
any
n the number of the process to be executed
where
`grd1`: $n \in 1..N \wedge a(n) = 1 \wedge toset(n) = 1$
the process is active and has not assigned a value to *out* yet
then
`act1`: $out : |(out \neq -1 \wedge out' = out) \vee (out = -1 \wedge (\exists nn \cdot nn \in 1..N \wedge out' = in(nn)))$
the assigned value equals $in(nn)$ of some nn, but nn is not necessarily n
`act2`: $toset(n) := 0$
set its sign to 0
end
Event *Decide* $\widehat{=}$
Status convergent

any
n the number of the process to be executed
where
grd1 : $n \in 1..N \wedge a(n) = 1 \wedge out \in 0..1$
the process is active and *out* has already been set
then
act1 : $d(n) := out$
set the decision to be *out*
act2 : $a(n) := 0$
no longer active
end
Event *PTerminates* $\widehat{=}$
Status convergent

any
n the number of the process to be terminated
where
grd1 : $n \in 1..N \wedge a(n) = 1$
the process is active
then
act1 : $a(n) := 0$
no longer active
end
VARIANT
$\mathtt{card}(\{\mathtt{n1} \cdot \mathtt{n1} \in 1..\mathtt{N} \wedge \mathtt{toset}(\mathtt{n1}) = 1 | \mathtt{n1}\}) +$
$\mathtt{card}(\{\mathtt{n2} \cdot \mathtt{n2} \in 1..\mathtt{N} \wedge \mathtt{a}(\mathtt{n2}) = 1 | \mathtt{n2}\})$
END

Adding restrictions.[2] Now only operations *read* and *write* on shared registers are atomic. The assignments $y[r_i] := v_i$ and $v_i := y[r_i]$ both have local variable v_i on one side, so both of them can be treated as atomic operations. Here we use this example to demonstrate the transformation $fwf(P)$. The additional guard of each event guarantees the existence of legal counter values and a new permutation p after its execution.

Again, termination is shown by counter *step*; its non-negativeness is guaranteed by proving the round counter r of each process is no more than $B+1$, therefore the step counter of each process is no more than $6*B+(B*(B+1))/2+5$, and the total step counter is no more than B times of this upper limit.

MACHINE DCFWF
Modelling distributed consensus, assuming that only operations *read* and *write* on shared registers are atomic
REFINES DC0
SEES Context
VARIABLES
a activeness of processes
out the output
d decisions of processes
y the preference of rounds
x the local preference coverage history of rounds
s state of processes
r local round numbers of processes
v local preferences of processes
j local loop variables of processes
B upper bound of times to delay any consecutively enabled event
C down-counters of processes
stepsums step counter-summations, $stepsums(n) - stepsums(n-1)$ is the step counter of the n-th process, and $stepsums(N)$ is the counter of total steps
the steps of event *PTerminates* are not counted, and the event does not affect the counters neither
s0 the number of total steps when the first time any process enters state 5 in the B-th round
INVARIANTS
inv1 : $y \in \mathbb{N}_1 \to -1..1$
$y(i)$ is the preference of the i-th round
inv2 : $x \in (\mathbb{N}_1 \times 0..1) \to 0..1$
$x(i,b) = 1$ means that at least one process n has entered state 2 in the i-th round, with local proference $v(n) = b$
inv3 : $s \in 1..N \to 1..7$
7 possible states for each process
inv4 : $r \in 1..N \to \mathbb{N}_1$
inv5 : $v \in 1..N \to 0..1$
inv6 : $j \in 1..N \to \mathbb{N}_1$
inv7 : $\forall n \cdot n \in 1..N \Rightarrow$
$((s(n) = 1 \wedge toset(n) = 1 \wedge r(n) > 1 \Rightarrow x((r(n)-1) \mapsto v(n)) = 1) \wedge$
$(s(n) \in 2..7 \Rightarrow x(r(n) \mapsto v(n)) = 1) \wedge$
$(s(n) \in 4..6 \Rightarrow y(r(n)) \in 0..1))$
some properties of x and y in different states;
used for proving validity
inv8 : $\forall n \cdot n \in 1..N \Rightarrow (\exists n1 \cdot n1 \in 1..N \wedge v(n) = in(n1))$
used for proving validity
inv9 : $\forall i \cdot i \in \mathbb{N}_1 \wedge y(i) \in 0..1 \Rightarrow (x(i \mapsto y(i)) = 1 \wedge (\exists n \cdot n \in 1..N \wedge y(i) = in(n)))$
relation between x and y, as well as and validity of $y(i)$ (preference of every round)
inv10 : $\forall i,b \cdot i \in \mathbb{N}_1 \wedge b \in 0..1 \wedge x(i \mapsto b) = 1 \Rightarrow (\forall ii \cdot ii \in 1..(i-1) \Rightarrow x(ii \mapsto b) = 1)$
if $x(i \mapsto b) = 1$ then b has left trace in x for all past rounds too
inv11 : $out \in 0..1 \Rightarrow (\exists n \cdot n \in 1..N \wedge$
$out = v(n) \wedge s(n) = 1 \wedge x(r(n) \mapsto out) = 1 \wedge x(r(n) \mapsto (1-out)) = 0)$
if *out* has been set then some process has left two key traces in x
inv12 : $\forall n \cdot n \in 1..N \wedge (out = -1 \vee (out = v(n) \wedge s(n) \in 2..4)) \Rightarrow$
$toset(n) = 1$
the relation among the states and the disappeared variable *toset*;
used for proving the guard refinement relationship of event *S4Y*
inv13 : $\forall n \cdot n \in 1..N \wedge s(n) = 4 \wedge x(r(n) \mapsto (1-v(n))) = 0 \Rightarrow (out = -1 \vee out = v(n))$
when *S4Y* is executed, either $out = -1$ or $out = v(n)$;
used for proving several POs of event *S4Y*
inv14 : $B \in \mathbb{N}_1 \wedge B \geq N$
B is no less than N

[2] Here is the transformed version in Event-B; a version of distributed consensus that uses "fair" events can be found in
`http://www.cas.mcmaster.ca/~zhangt26/ICTAC/appendix.pdf`

inv15 : $C \in 1\,..\,N \to 1\,..\,B$

inv16 : $stepsums \in 0\,..\,N \to \mathbb{N}$

inv17 : $\exists p \cdot p \in 1\,..\,N \rightarrowtail\!\!\!\!\!\twoheadrightarrow 1\,..\,N \wedge (\forall n \cdot n \in 1\,..\,N \Rightarrow C(p(n)) \geq n)$
such p, a permutation of $1\,..\,N$, always exists, that $C(p(n)) \geq n$ for $n \in 1\,..\,N$

inv18 : $\exists n0 \cdot n0 \in 1\,..\,N \wedge a(n0) = 1 \Rightarrow (\exists n \cdot n \in 1\,..\,N \wedge a(n) = 1 \wedge (\exists CC \cdot CC = (1\,..\,N \times \{B\}) \lhd\!\!+ (\lambda nn \cdot nn \in 1\,..\,N \setminus \{n\} \wedge a(nn) = 1 | C(nn) - 1) \wedge CC \in 1\,..\,N \to 1\,..\,B \wedge (\exists p \cdot p \in 1\,..\,N \rightarrowtail\!\!\!\!\!\twoheadrightarrow 1\,..\,N \wedge (\forall nnn \cdot nnn \in 1\,..\,N \Rightarrow CC(p(nnn)) \geq nnn))))$
such a p prevents unexpected termination caused by restriction on counters, i.e., the additional guards are always satisfiable as long as the original guards are satisfiable;
when $\exists n0 \cdot n0 \in 1\,..\,N \wedge a(n0) = 1$, suppose $mn = min(\{n | n \in 1..N \wedge a(p(n)) = 1\})$, then with $n = p(mn)$ and $p = ((\lambda n \cdot n \in 1\,..\,N - mn | p(n + mn)) \cup (\lambda n \cdot n \in N + 1 - mn\,..\,N | p(n + mn - N))$, CC satisfies the invariant in the next state

inv19 : $\forall n \cdot n \in 1\,..\,N \wedge a(n) = 1 \Rightarrow stepsums(N) \leq B * (stepsums(n) - stepsums(n-1) + 1) - C(n)$
the number of total steps are no more than B times of the step counter of any active process

inv20 : $stepsums(0) = 0 \wedge (\forall n \cdot n \in 1\,..N \Rightarrow ((s(n) = 1 \wedge out = -1 \Rightarrow stepsums(n) - stepsums(n-1) \leq 6 * (r(n) - 1) + (r(n) * (r(n) - 1))/2 + 2 - a(n)) \wedge (s(n) = 1 \wedge out \neq -1 \Rightarrow stepsums(n) - stepsums(n-1) \leq 6 * (r(n) - 1) + (r(n) * (r(n) - 1))/2 + 6 - a(n)) \wedge (s(n) \in 2\,..\,4 \Rightarrow stepsums(n) - stepsums(n-1) \leq 6*(r(n)-1)+(r(n)*(r(n)-1))/2+s(n)) \wedge (s(n) = 5 \Rightarrow stepsums(n) - stepsums(n-1) \leq 6 * (r(n) - 1) + (r(n) * (r(n) - 1))/2 + 4 + j(n)) \wedge (s(n) = 6 \Rightarrow stepsums(n) - stepsums(n-1) \leq 6*(r(n)-1)+(r(n)*(r(n)+1))/2+5) \wedge (s(n) = 7 \Rightarrow stepsums(n) - stepsums(n-1) \leq 6*(r(n)-1)+(r(n)*(r(n)+1))/2+6)))$
the upper bound on step counter of each process at each state

inv21 : $s0 \in \mathbb{N}$

inv22 : $s0 = 0 \Rightarrow (\forall n \cdot n \in 1\,..\,N \Rightarrow r(n) < B \vee (r(n) = B \wedge s(n) \leq 4))$
$s0$ is 0 until some process enters state 5 in the B-th round

inv23 : $s0 > 0 \Rightarrow (y(B) \in 0\,..\,1 \wedge stepsums(N) \geq s0 \wedge (\forall n \cdot n \in 1\,..\,N \Rightarrow ((a(n) = 1 \wedge r(n) = B \wedge s(n) = 3 \Rightarrow C(n) \leq B - stepsums(N) + s0) \wedge (stepsums(N) < s0 + B \Rightarrow ((r(n) < B \vee (r(n) = B \wedge s(n) \leq 5)) \wedge (a(n) = 1 \wedge r(n) = B \wedge s(n) = 5 \Rightarrow j(n) \leq stepsums(N) - s0 + 1))))))$
some properties when $s0 > 0$, and when $s0 > 0 \wedge stepsums(N) < s0 + B$;
used for proving that when any process is in state 6 of the B-th round, $stepsums(N) \geq s0 + B$ and at that time, no process is in state 3 of the B-th round (and no process can enter anymore)

inv24 : $\forall n \cdot n \in 1\,..\,N \wedge ((r(n) = B \wedge s(n) = 7) \vee (r(n) = B + 1 \wedge s(n) \in 1\,..\,4)) \Rightarrow v(n) = y(B)$
once a process reaches state 7 in the B-th round, its local preference will equal $y(B)$ until it terminates, and $y(B)$ will not vary due to *inv*23, because $stepsums(N) \geq s0 + B$; thus no process is in state 3 nor can any process enter state 3 to set $y(B)$;
used for proving *inv*25

inv25 : $((s0 = 0 \vee (s0 > 0 \wedge stepsums(N) < s0 + B)) \Rightarrow (x(B + 1 \mapsto 0) = 0 \wedge x(B + 1 \mapsto 1) = 0)) \wedge (s0 > 0 \wedge stepsums(N) \geq s0 + B \Rightarrow x(B + 1 \mapsto 1 - y(B)) = 0)$
$x(B + 1 \mapsto 1 - y(B)) = 0$ when $s0 > 0$;
used for proving *inv*26

inv26 : $\forall n \cdot n \in 1\,..\,N \Rightarrow (r(n) \leq B \vee (r(n) = B + 1 \wedge s(n) \in 1\,..\,4))$
no process can enter state 5 in the $(B + 1)$-th round;
this puts an upper bound on the step counter of single processes and thus on total steps, which is then used to prove termination (non-negativeness of the variant)

EVENTS

Initialisation

begin

act1 : $a := 1\,..\,N \times \{1\}$

act2 : $out := -1$

act3 : $d := 1\,..\,N \times \{-1\}$

act4 : $y := \mathbb{N}_1 \times \{-1\}$

act5 : $x := (\mathbb{N}_1 \times 0\,..\,1) \times \{0\}$

act6 : $s := 1\,..\,N \times \{1\}$

act7 : $r := 1\,..\,N \times \{1\}$

act8 : $v := in$

act9 : $j := 1\,..\,N \times \{1\}$

act10 : $B, C : |B' \in \mathbb{N}_1 \wedge B' \geq N \wedge C' = 1\,..\,N \times \{B'\}$

act11 : $stepsums := 0\,..\,N \times \{0\}$

act12 : $s0 := 0$

end

Event *S1* $\widehat{=}$
enter the $r(n)$-th round and leave a record of local preference in x

Status convergent

any

n the number of the process to be executed (the same meaning in all following events except *PTerminates*)

CC the value of C after execution (the same meaning in all following events except *PTerminates*)

where

grd1 : $n \in 1\,..\,N \wedge a(n) = 1 \wedge s(n) = 1$
the process is active and in state 1

grd2 : $out = -1$
out has not been set

grd3 : $CC = (1\,..\,N \times \{B\}) \lhd\!\!+ (\lambda nn \cdot nn \in 1\,..\,N \setminus \{n\} \wedge a(nn) = 1 | C(nn) - 1) \wedge CC \in 1\,..\,N \to 1\,..\,B \wedge (\exists p \cdot p \in 1\,..\,N \rightarrowtail\!\!\!\!\!\twoheadrightarrow 1\,..\,N \wedge (\forall nnn \cdot nnn \in 1\,..\,N \Rightarrow CC(p(nnn)) \geq nnn))$
CC sets the counters of n and inactive processes to be B, decreases the counters of other processes by 1, and a new permutation exists (the same meaning in all following events except *PTerminates*)

then

act1 : $x(r(n) \mapsto v(n)) := 1$

act2 : $s(n) := 2$
set the state to be 2

act3 : $C := CC$
update the counters (the same meaning in all following events except *PTerminates*)

act4 : $stepsums := stepsums \lhd\!\!+ (\lambda nn \cdot nn \in n\,..\,N | stepsums(nn) + 1)$
update the counter-summations (the same meaning in all following events except *PTerminates*)

end
Event *S2Y* $\widehat{=}$
if the preference of the $r(n)$-th round has not been set, try to set it to be local preference
Status convergent
any
n
CC
where
grd1 : $n \in 1..N \wedge a(n) = 1 \wedge s(n) = 2 \wedge y(r(n)) = -1$
the process is active and in state 2, the preference of the $r(n)$-th round has not been set
grd2 : $CC = (1..N \times \{B\}) \lhd (\lambda nn \cdot nn \in 1..N \setminus \{n\} \wedge a(nn) = 1 | C(nn) - 1) \wedge CC \in 1..N \to 1..B \wedge (\exists p \cdot p \in 1..N \rightarrowtail\!\!\!\!\!\!\rightarrow 1..N \wedge (\forall nnn \cdot nnn \in 1..N \Rightarrow CC(p(nnn)) \geq nnn))$
then
act1 : $s(n) := 3$
set the state to be 2
act2 : $C := CC$
act3 : $stepsums := stepsums \lhd (\lambda nn \cdot nn \in n..N | stepsums(nn) + 1)$
end
Event *S2N* $\widehat{=}$
...
⋮
Event *S7* $\widehat{=}$
...[3]
Event *Decide* $\widehat{=}$
refines *Decide*
any
n
CC
where
grd1 : $n \in 1..N \wedge a(n) = 1 \wedge s(n) = 1$
the process is active and in state 1
grd2 : $out \in 0..1$
out has already been set
grd3 : $CC = (1..N \times \{B\}) \lhd (\lambda nn \cdot nn \in 1..N \setminus \{n\} \wedge a(nn) = 1 | C(nn) - 1) \wedge CC \in 1..N \to 1..B \wedge (\exists p \cdot p \in 1..N \rightarrowtail\!\!\!\!\!\!\rightarrow 1..N \wedge (\forall nnn \cdot nnn \in 1..N \Rightarrow CC(p(nnn)) \geq nnn))$
then
act1 : $d(n) := out$
set the decision to be out
act2 : $a(n) := 0$
no longer active
act3 : $C := CC$
act4 : $stepsums := stepsums \lhd (\lambda nn \cdot nn \in n..N | stepsums(nn) + 1)$
end
Event *PTerminates* $\widehat{=}$
refines *PTerminates*
any
n number of the process to be terminated
where
grd1 : $n \in 1..N \wedge a(n) = 1$
the process is active
then
act1 : $a(n) := 0$
no longer active
end
VARIANT
$B * (6 * B + (B * (B + 1))/2 + 5) - \mathtt{stepsums(N)}$
END

7 Conclusions

This work started with the goal of expressing action systems with fairness in formalisms like Event-B. The core is the observation that a modification of the transformation $fin(P)$ of [3] from standard transition systems to a finitary weakly fair one is suitable. It was shown that that all finitary weakly fair computations of P terminate, if and only if all computations of $fin(P)$ terminate. However, the schedules of $fin(P)$ are not exactly the finitary weakly fair ones of P, because the termination in some computations may be caused by improper scheduling: $fin(P)$ may reach a state q', such that in its corresponding state q of P, some transitions are enabled, but transitions of $fin(P)$ are disabled by the additional guards that restrict the counters. This inequality does not affect the correctness of the proof mentioned above, but this type of termination makes it unsuitable to model practical transition systems, because the system after transformation will be at risk of terminating unexpectedly. Certain temporal logic properties cannot be proved, due to the difficulty of distinguishing terminations caused by original guards or by a counter of value 0. The transformation $fwf(P)$ and $dist(P)$ suggested here guarantee that the schedules remain equivalent. Thus, lower bound of steps can be easily proved, and all temporal logic properties are preserved. We have demonstrated the application of $dist(P)$ with the development of the alternating bit protocol and the application of

[3] Events *S2N* to *S7* are omitted here since they are all similarly translated atomic operations, refer to `http://www.cas.mcmaster.ca/~zhangt26/ICTAC/appendix.pdf` for the complete code.

$fwf(P)$ with the development of distributed consensus. We believe that this is the first mechanically checked development of this distributed consensus algorithm.

In this paper, we have considered only weak fairness. A similar transformation for finitary strong fairness waits to be worked out. While the examples of the paper have been processed with Rodin [4], the transformation was done by hand, and the proof was semi-automatic. Verifying the distributed consensus model with finitary fairness is more time-consuming than verifying the ABP model, because modelling an arbitrary number of counters requires additional functions to express the restrictions. It would be useful to automate this transformation, as well as the verification of the proof obligations that only involve the bound and counters, which is irrelevant to the original model.

Acknowledgement. We thank the reviewers for their helpful comments.

References

1. Back, R.J.R.: Refinement calculus, part II: Parallel and reactive programs. In: de Bakker, J.W., de Roever, W.-P., Rozenberg, G. (eds.) REX 1989. LNCS, vol. 430, pp. 67–93. Springer, Heidelberg (1990)
2. Abrial, J.R.: Modeling in Event-B: System and Software Engineering. Cambridge University Press (2010)
3. Alur, R., Henzinger, T.A.: Finitary fairness. ACM Trans. Program. Lang. Syst. 20(6), 1171–1194 (1998)
4. Abrial, J.-R., Butler, M., Hallerstede, S., Voisin, L.: An open extensible tool environment for event-B. In: Liu, Z., Kleinberg, R.D. (eds.) ICFEM 2006. LNCS, vol. 4260, pp. 588–605. Springer, Heidelberg (2006)
5. Chandy, K.M., Misra, J.: Parallel Program Design: A Foundation. Addison-Wesley (1988)
6. Francez, N.: Fairness. Texts and Monographs in Computer Science. Springer (1986)
7. Lamport, L.: The temporal logic of actions. ACM Transactions on Programming Languages and Systems 16(3), 872–923 (1994)
8. Back, R., Xu, Q.: Refinement of fair action systems. Acta Informatica 35(2), 131–165 (1998)
9. Singh, A.K.: Program refinement in fair transition systems. Acta Informatica 30, 503–535 (1993)
10. Wabenhorst, A.: Stepwise development of fair distributed systems. Acta Informatica 39, 233–271 (2003)
11. Apt, K.R., Olderog, E.R.: Proof rules and transformations dealing with fairness. Sci. Comput. Program. 3(1), 65–100 (1983)
12. Fischer, M., Lynch, N., Paterson, M.: Impossibility of distributed consensus with one faulty process. Journal of the ACM 32, 374–382 (1985)
13. Bartlett, K.A., Scantlebury, R.A., Wilkinson, P.T.: A note on reliable full-duplex transmission over half-duplex links. Communications of the ACM 12(5), 260–261 (1969)
14. Wabenhorst, A.: A stepwise development of the alternating bit protocol. Technical Report PRG-TR-12-97, Oxford University Computing Laboratory (March 1997)
15. Feijen, W.H.J., van Gasteren, A.J.M.: On a Method of Multiprogramming. Springer (1999)
16. Sekerinski, E.: An algebraic approach to refinement with fair choice. Electronic Notes in Theoretical Computer Science 214, 51–79 (2008)