Operational Semantics of Cooperative Fair Threads

Frédéric Boussinot*
Frederic.Boussinot@sophia.inria.fr

June 2002

Abstract

Fair threads are cooperative threads run by a fair scheduler which gives them equal access to the processor. Fair threads can communicate by broadcast events. The paper describes formal semantics for fair threads and fair schedulers in the structural operational semantics format.

Keywords: Concurrency, Threads, Cooperative Scheduling, Structural Operational Semantics

1 Introduction

FairThreads is a new framework for concurrent programming which puts the focus on simplicity, clarity, and portability. The intension is to provide users with an alternative to standard approaches of multithreading such as POSIX Pthreads in C, or Java threads.

FairThreads has been first proposed in the context of Java[2]. It has then been introduced in the Bigloo[8] implementation of Scheme. Recently, a version of FairThreads has been proposed for C[9]. FairThreads definition is relying on previous work belonging to the so-called reactive approach described in the Web site [7]. FairThreads has strong links with the Junior framework[4, 5] which is a set of Java classes for reactive programming. Comparison of Java threads and SugarCubes, a framework closely related to Junior, can be found in [1].

In order to get a simple and clear framework, it appears that a formal approach is mandatory. The purpose of this paper is to describe the formal operational semantics of a large fragment of FairThreads. More precisely, one considers the cooperative part of the framework, that will be called Cooperative FairThreads, and gives rewriting rules for it. This semantics is close to the one of SugarCubes, described in [1]. A denotational semantics is presently under work for the version of Cooperative FairThreads implemented in Bigloo.

Cooperative FairThreads

Cooperative fair threads are cooperative threads run by a scheduler which gives them equal access to the processor. As usual in cooperative contexts, fair threads must never forget to cooperate with other threads. Scheduler fairness has two aspects:

- Fair schedulers define execution *instants* in which all started threads run up to their next cooperation point.

*EMP/CMA-INRIA, MIMOSA project.

- Fair schedulers dispatch the same information to all threads. More precisely, events generated in a scheduler are broadcast to all the threads it schedules.

Cooperative frameworks are generally considered to be simpler than preemptive ones. Indeed, as preemption cannot occur in an uncontroled way, cooperative frameworks are less undeterministic. Actually, Cooperative FairThreads puts the situation to an extreme point, as it is fully deterministic; threads are chosen for execution following a strict round-robin algorithm. This can be a great help in programming and debugging.

Fairness in event processing means that all threads waiting for an event receive it during the very instant it is generated; thus, a thread leaving control to cooperate with othes threads does not risk to loose an event generated later in the same instant. Note that scheduler instants define time scopes of events.

Cooperative FairThreads is fully portable as its semantics does not depends on the executing platform.

Structure of the paper

The paper has the following structure: section 2 describes the syntax considered and the notations used; section 3 gives an intuitive overview of the semantics; rewriting rules for instructions used by fair threads are described in section 4; finally, section 5 describes semantics of fair threads and of fair schedulers.

2 Syntax and Notations

As FairThreads are used in several different contexts, one defines an abstract syntax which can be easily translated in the concrete syntax used in these contexts. Moreover, as the goal is to give semantics for Cooperative FairThreads, not all primitives of FairThreads are considered, but only those that fit in the purely cooperative framework (for example, unlinked threads which are defined in the C version of FairThreads are not considered because they are basically preemptive).

2.1 Threads

A fair thread t is basically made of 3 components:

- $t.run$ is the instruction run by the thread.
- $t.status$ is the execution status; initially, it has value $CONT$, which means that execution has to be continued during the current instant. Status is $TERM$ when the thread is totally terminated; it is $COOP$ when the thread cooperates because it has finished execution for the current instant.
- $t.susp$ is a boolean which is true if the thread is suspended, and false otherwise.

One associates to each thread t a special event, denoted by $term(t)$, used to signal the total termination of t.

2.2 Instructions

The syntax of instructions is defined by:

$$\begin{array}{rcl} inst & ::= & basic\_inst \\ & | & sync\_inst \\ & | & control\_inst \\ & | & timed\_inst \\ & | & get\_value\_inst \end{array}$$

Basic Instructions

Amongst basic instructions, are calls, sequence of instructions, while loops, boolean tests, etc.

$$
\begin{array}{lcl}
basic\_inst & ::= & call \\
 & | & inst_1 \ ... \ inst_n \\
 & | & while(exp) \ inst \\
 & | & if(exp) \ inst_1 \ else \ inst_2 \\
 & | & ...
\end{array}
$$

Calls are means to change the environment; their exact syntax does not matter for the abstract syntax (in concrete syntax, calls can be function, procedure, or method calls).

Synchronizing Instructions

Instructions for synchronization are: cooperate, waiting for an event, generation of an event, and joining a thread (that is, waiting for termination of it).

$$
\begin{array}{lcl}
sync\_inst & ::= & cooperate \\
 & | & await(event) \\
 & | & generate(event) \\
 & | & generate\_value(event, value) \\
 & | & join(thread)
\end{array}
$$

Control Instructions

Control instructions give fine control over threads: basically, threads can be created, stopped, suspended, and resumed.

$$
\begin{array}{lcl}
control\_inst & ::= & create(thread) \\
 & | & stop(thread) \\
 & | & suspend(thread) \\
 & | & resume(thread)
\end{array}
$$

Timed Instructions

Timed instructions terminate when a delay (actually, defined by a number of instants) expired; cooperate, await, and join are concerned.

$$
\begin{array}{lcl}
timed\_inst & ::= & cooperate\_n(number) \\
 & | & await\_n(event, number) \\
 & | & join\_n(thread, number)
\end{array}
$$

Get Value Instruction

The *get_value* instruction returns the kth values (it it exists) associated to a given event.

$$
get\_value\_inst \ ::= \ get\_value(thread, event, k)
$$

Correspondance with the API in C

The correspondance with the C API of [9] is straightforward:

- *cooperate*: `ft_thread_cooperate`
- *await*: `ft_thread_await`
- *generate*: `ft_thread_generate`
- *generate_value*: `ft_thread_generate_value`
- *join*: `ft_thread_join`
- *create*: `ft_thread_create`
- *stop*: `ft_scheduler_stop`
- *suspend*: `ft_scheduler_suspend`
- *resume*: `ft_scheduler_resume`
- *cooperate_n*: `ft_thread_cooperate_n`
- *await_n*: `ft_thread_await_n`
- *join_n*: `ft_thread_join_n`
- *get_value*: `ft_thread_get_value`

Note that only one scheduler is considered in Cooperative FairThreads (it can thus be considered as implicit). For simplicity, error codes returned by instructions are not considered.

2.3 Schedulers

A fair scheduler *sched* is made of several components:

- *sched.actual* is the list of active threads.
- *sched.events* is the list of generated events.
- *sched.eoi* is a boolean which is set to true at the end of each instant.
- *sched.move* is a boolean which is set to true when a new event is generated.
- *sched.to_broadcast* is the list of events to be considered as present at next instant.
- *sched.to_start* is the list of threads to be started at next instant.
- *sched.to_stop* is the list of threads to be stopped at next instant.
- *sched.to_suspend* is the list of threads to be suspended at next instant.
- *sched.to_resume* is the list of threads to be resumed at next instant.
- *sched.values* is the list of values associated to generated events.

2.4 Notations

Fair threads semantics is expressed in the so-called structural operational semantics (SOS) format defined in [6].

Expressions

Evaluation of expression exp in the environment env is written:

$$exp \models env \xrightarrow{v} env'$$

v is the result of evaluation, and env' is the resulting environment.

Instructions

Execution of the instruction $inst$ by the scheduler $sched$ in the environment env is written:

$$inst, sched, env \xrightarrow{\alpha} inst', sched', env'$$

$inst'$ is the resulting instruction, and α is the status ($TERM$, $COOP$, $CONT$) after execution; $sched'$ is the resulting scheduler and env' is the resulting environment.

Threads

Execution of the thread t by the scheduler $sched$ in the environment env is written:

$$t, sched, env \xrightarrow{b} t', sched', env'$$

t' is the thread after execution, and b is a boolean which is true if execution is finished for the current instant; $sched'$ is the resulting scheduler and env' is the resulting environment.

Schedulers

One execution step of the scheduler $sched$ with the environment env is written:

$$sched, env \xrightarrow{b} sched', env'$$

$sched'$ is the resulting scheduler and env' is the resulting environment; the boolean b is true if all threads have finished their execution for the current instant.

A sequence of execution steps leading to a situation where all threads are finished for the current instant defines a complete instant of $sched$; it is written:

$$sched, env \Longrightarrow sched', env'$$

$sched'$ is the resulting scheduler and env' is the resulting environment.

3 Overview of the Semantics

In this section, one gives an intuitive overview of the semantics.

3.1 Instants

Let's consider a scheduler $sched$ and an environment env. The final goal of the semantics is to shows how $sched$ and env evolves as time passes. Actually, evolution is decomposed in a sequence of instants; first instant is:

$$sched, env \Longrightarrow sched_1, env_1$$

which means that, starting from $sched$ and env, one gets $sched_1$ and env_1 at the end of the instant.

Threads are not immediately started as soon as they are created in order to avoid interferences with currently running threads. Actually, all threads created during one instant are stored in $sched.to\_start$, and are actually started at the beginning of the next instant. In the same way, events broadcast from the outside world are stored in $sched.to\_broadcast$ and are incorporated in the system at the beginning of the next instant. Stop, suspend, and resume orders are processed in the same way.

Thus, evolution of the scheduler as time passes is a sequence of the form:

$$sched, env \Longrightarrow sched_1, env_1 \quad sched'_1, env_1 \Longrightarrow sched_2, env_2 \quad sched'_2, env_2 \Longrightarrow sched_3, env_3 \; ...$$

where $sched'_i = sched_i$ except that all orders collected during instant i are incorporated in $sched'_i$.

Now, let's decompose instants: each instant consists in cyclically running all threads that are to be continued. Running a thread t means to resume execution of its associated instruction $t.run$. A thread which is to be continued is a thread which is neither suspended nor completely terminated, and which has not already cooperated during the instant. The instant is over when all threads that are not suspended are either terminated or have cooperated.

Threads which are started are placed in the *actual* vector of the scheduler. At the beginning of each instant, threads that are completely terminated ($TERM$) and threads stopped during previous instant are removed from *actual*. Remaining threads receives the status $CONT$ meaning that they are to be continued. Thus, at the beginning of each instant, all threads in *actual* are either suspended or are to be continued. At the end of the instant, all threads are either suspended, or terminated ($TERM$), or have cooperated ($COOP$). Note that the order in which threads are executed always remains the same during an instant. Note also that suspensions and resumptions of threads do no change the order in which the other threads are executed.

3.2 Instructions

An execution step of an instruction *inst* run by a scheduler *sched* in an environment *env* is written:

$$inst, sched, env \xrightarrow{\alpha} inst', sched', env'$$

inst is the instruction associated to the thread t ($t.run$) started in *sched*, and α is the code returned after execution:

- Code $TERM$ means that t is completely terminated, and, thus, must be removed from the scheduler.
- Code $COOP$ means that t cooperates, having finished to execute for the current instant; in this case, $inst'$ is the new instruction that t has to run at the next instant. Basically, $COOP$ is produced by the *cooperate* instruction.
- Code $CONT$ means that t must be continued during the current instant; actually, this happens when t awaits an event e which is not present; in this case, the scheduler gives control to the others threads which can generate e; as t is to be continued for the current instant, the scheduler will resume it; thus, if e is finally generated, it will be seen by t.

The scheduler behavior consists in cyclically running the threads in *actual* until some have to be continued. The number of cycles depends on the generated events. For example, consider a situation where $actual =< t_1, t_2 >$ and t_1 awaits the event e generated by t_2. Then, after t_2 execution, a new cycle is needed to resume t_1. Otherwise, t_1 would not consider e as present despite the fact that it is generated; in such a situation, one could not say that e is broadcast. Note that a third cycle would be necessary if, after generating e, t_2 where waiting for another event generated by t_1. Actually, new cycles are needed until one reaches a situation where no new event is generated; then, the end of the current instant can be safely decided.

The scheduler uses two boolean flags to manage instants:

- The flag *move* is set each time a new event is generated; it is reset by the scheduler at the beginning of each cycle. The scheduler does not decide the end of the current instant when *move* is set, to give threads waiting for some generated event the possibility to react to it.

- The flag *eoi* is set by the scheduler when the end of the current instant is decided; it is reset at the beginning of each new instant. It is used to inform threads which are waiting for events that these events are definitely absent. Then, the threads cooperate, which leads to a situation where all non-suspended threads in *actual* are terminated, or have cooperated. At that point, the next instant can safely take place.

Awaiting an event which is absent blocks a thread up to the end of the current instant. This forbids immediate (that is, during same instant) reaction to the absence of an event; reaction, if any, is thus postponed to next instant. This is important to avoid situations where one could react to the absence of an event during a instant by generating it during the same very instant, which would contradict the fact that the event is absent. These kind of contradictions, known as "causality problems" in synchronous languages[3], do not exist with fair threads. In the same way, trying to get a not available generated value blocks a thread up to the end of the current instant.

4 Instructions

This section describes the rewriting rules defining the semantics of instructions.

4.1 Call

A call immediately terminates after running the called function:

$$call, sched, env \xrightarrow{TERM} nothing, sched, env' \tag{1}$$

where env' is the environment obtained after executing *call* in *env*, and *nothing* is the instruction that does nothing:

$$nothing, sched, env \xrightarrow{TERM} nothing, sched, env \tag{2}$$

4.2 Sequence

For simplicity, one only considers binary sequences (general sequences can be coded with binary ones). There are two rules, depending on the termination of the first branch. If the first branch terminates, then the second one is immediately run:

$$\frac{inst_1, sched, env \xrightarrow{TERM} inst_1', sched', env' \quad inst_2, sched', env' \xrightarrow{\alpha} inst_2', sched'', env''}{inst_1 \; inst_2, sched, env \xrightarrow{\alpha} inst_2', sched'', env''} \tag{3}$$

If the first branch is not terminated, then so is the sequence:

$$\frac{inst_1, sched, env \xrightarrow{\alpha} inst_1', sched', env' \quad \alpha \neq TERM}{inst_1 \; inst_2, sched, env \xrightarrow{\alpha} inst_1' \; inst_2, sched', env'} \tag{4}$$

4.3 While Loop

A while loop tests a boolean condition and terminates immediately if it is false:

$$\frac{exp \models env \xrightarrow{false} env'}{while(exp) \; inst, sched, env \xrightarrow{TERM} nothing, sched, env'} \tag{5}$$

The body is executed when the condition is true:

$$\frac{exp \models env \xrightarrow{true} env' \quad while_{inst}(exp) \; inst, sched, env' \xrightarrow{\alpha} inst', sched', env''}{while(exp) \; inst, sched, env \xrightarrow{\alpha} inst', sched', env''} \tag{6}$$

The auxiliary $while_{inst}$ instruction is defined in the following section.

Auxiliary While Instruction

The auxiliary while instruction runs the loop body up to termination, without re-evaluating the boolean condition; then, it rewrites as a standard while loop which evaluates the condition to determine if the body has to be run.

$$\frac{inst, sched, env \xrightarrow{TERM} inst', sched', env' \quad while(exp)\ initial, sched', env' \xrightarrow{\alpha} inst'', sched'', env''}{while_{initial}(exp)\ inst, sched, env \xrightarrow{\alpha} inst'', sched'', env''} \tag{7}$$

$$\frac{inst, sched, env \xrightarrow{\alpha} inst', sched', env' \quad \alpha \neq TERM}{while_{initial}(exp)\ inst, sched, env \xrightarrow{\alpha} while_{initial}(exp)\ inst', sched', env'} \tag{8}$$

4.4 If

The left branch is chosen if a boolean condition is true:

$$\frac{exp \models env \xrightarrow{true} env' \quad inst_1, sched, env' \xrightarrow{\alpha} inst'_1, sched', env''}{if(exp)\ inst_1\ else\ inst_2, sched, env \xrightarrow{\alpha} inst'_1, sched', env''} \tag{9}$$

Otherwise, the right branch is executed:

$$\frac{exp \models env \xrightarrow{false} env' \quad inst_2, sched, env' \xrightarrow{\alpha} inst'_2, sched', env''}{if(exp)\ inst_1\ else\ inst_2, sched, env \xrightarrow{\alpha} inst'_2, sched', env''} \tag{10}$$

4.5 Cooperate

The *cooperate* statement finishes execution for the current instant; moreover, nothing remains to be done at the next instant (thus, execution at the next instant will resume in sequence from *cooperate*):

$$cooperate, sched, env \xrightarrow{COOP} nothing, sched, env \tag{11}$$

4.6 Generate

A *generate* statement adds the generated event in the event set of the scheduler, and immediately terminates; moreover, the *move* flag is set to indicate that something new happened:

$$generate(event), sched, env \xrightarrow{TERM} nothing, sched', env \tag{12}$$

where $sched' = sched[events \mathrel{+}= event][move := true]$.

If a value is associated to the generation, it is added at the end of the table of values associated to the event.

$$generate(event, v), sched, env \xrightarrow{TERM} nothing, sched', env \tag{13}$$

where $sched' = sched[events \mathrel{+}= event][move := true][values(event) \mathrel{+}= v]$.

4.7 Await

An instruction *await* terminates immediately if the awaited event is present:

$$\frac{event \in sched.events}{await(event), sched, env \xrightarrow{TERM} nothing, sched, env} \tag{14}$$

An instruction *await* has to be continued if the awaited event is not generated while the current instant is not terminated:

$$\frac{event \notin sched.events \quad sched.eoi = false}{await(event), sched, env \xrightarrow{CONT} await(event), sched, env} \tag{15}$$

An instruction *await* cooperates if the awaited event is absent, that is, it is not generated and the current instant is terminated:

$$\frac{event \notin sched.events \quad sched.eoi = true}{await(event), sched, env \xrightarrow{COOP} await(event), sched, env} \tag{16}$$

4.8 Join

Nothing is done if the thread to be joined is already terminated:

$$\frac{t.status = TERM}{join(t), sched, env \xrightarrow{TERM} nothing, sched, env} \tag{17}$$

If the thread is not already terminated, semantics of join is to wait for the event generated by the scheduler when the thread terminates (see section 5.1).

$$\frac{t.status \neq TERM \quad await(term(t)), sched, env \xrightarrow{\alpha} inst, sched', env'}{join(t), sched, env \xrightarrow{\alpha} inst, sched', env'} \tag{18}$$

4.9 Create

Execution of $create(t)$ adds t to the vector *to_start*. Thus, t will be started at the next instant.

$$create(t), sched, env \xrightarrow{TERM} nothing, sched[to\_start \mathrel{+}= t], env \tag{19}$$

4.10 Stop

Execution of $stop(t)$ adds t to the vector *to_stop*. Thus, t will be stopped at the next instant.

$$stop(t), sched, env \xrightarrow{TERM} nothing, sched[to\_stop \mathrel{+}= t], env \tag{20}$$

4.11 Suspend

Execution of $suspend(t)$ adds t to the vector *to_suspend*. Thus, t will be suspended at the next instant.

$$suspend(t), sched, env \xrightarrow{TERM} nothing, sched[to\_suspend \mathrel{+}= t], env \tag{21}$$

4.12 Resume

Execution of $resume(t)$ adds t to the vector to_resume. Thus, t will be resumed at next instant.

$$resume(t), sched, env \stackrel{TERM}{\longrightarrow} nothing, sched[to\_resume \mathrel{+}= \ t], env \tag{22}$$

4.13 Cooperate_n

Execution of $cooperate\_n(k)$ has no effect if the delay defined by k is expired:

$$\frac{k \leq 0}{cooperate\_n(k), sched, env \stackrel{TERM}{\longrightarrow} nothing, sched, env} \tag{23}$$

Otherwise, the thread cooperates and the instruction to be executed at next instant is $cooperate\_n(k-1)$.

$$\frac{k > 0}{cooperate\_n(k), sched, env \stackrel{COOP}{\longrightarrow} cooperate\_n(k-1), sched, env} \tag{24}$$

Actually, $cooperate\_n(k)$ is equivalent to the loop $for(int\ i = 0; i < k; i++)\ cooperate()$.

4.14 Await_n

Execution of $await\_(e, k)$ terminates immediately if e is present or if the delay defined by k is expired:

$$\frac{event \in sched.events \quad or \quad k \leq 0}{await\_n(event, k), sched, env \stackrel{TERM}{\longrightarrow} nothing, sched, env} \tag{25}$$

Execution of $await\_(e, k)$ is to be continued if e is not present while the current instant is not terminated:

$$\frac{event \notin sched.events \quad k > 0 \quad sched.eoi = false}{await\_n(event, k), sched, env \stackrel{CONT}{\longrightarrow} await\_n(event, k), sched, env} \tag{26}$$

Execution of $await\_n(e, k)$ cooperates if e is absent; moreover, the instruction to be executed at the next instant is $await\_n(e, k-1)$:

$$\frac{event \notin sched.events \quad k > 0 \quad sched.eoi = true}{await\_n(event, k), sched, env \stackrel{COOP}{\longrightarrow} await\_n(event, k-1), sched, env} \tag{27}$$

4.15 Join_n

Nothing is to be done if the joined thread is already terminated or if the delay is expired:

$$\frac{t.status = TERM \quad or \quad k \leq 0}{join\_n(t, k), sched, env \stackrel{TERM}{\longrightarrow} nothing, sched, env} \tag{28}$$

Otherwise, the semantics is defined, using $await\_n$, by:

$$\frac{t.status \neq TERM \quad await\_n(term(t), k), sched, env \stackrel{\alpha}{\longrightarrow} inst, sched', env'}{join\_n(t, k), sched, env \stackrel{\alpha}{\longrightarrow} inst, sched', env'} \tag{29}$$

4.16 Generated Values

If a value is available then it is returned and execution terminates immediately:

$$\frac{sched.values(event).length \geq k}{get\_value(t, event, k), sched, env \overset{TERM}{\longrightarrow} nothing, sched, env'} \tag{30}$$

where env' is env, transformed by the return of the value (this is not modelized here).

Execution is to be continued if no value is available while current instant is not terminated:

$$\frac{sched.values(event).length < k \quad sched.eoi = false}{get\_value(t, event, k), sched, env \overset{CONT}{\longrightarrow} get\_value(t, event, k), sched, env} \tag{31}$$

If no value is available when the current instant is over, then the instruction simply cooperates:

$$\frac{sched.values(event).length < k \quad sched.eoi = true}{get\_value(t, event, k), sched, env \overset{COOP}{\longrightarrow} nothing, sched, env} \tag{32}$$

5 Threads and Schedulers

This section describes the rewiting rules defining the semantics of threads and schedulers.

5.1 Thread

Nothing is done for a thread which is suspended, or whose status is different from $CONT$:

$$\frac{t.susp = true \quad or \quad t.status \neq CONT}{t, sched, env \overset{true}{\longrightarrow} t, sched, env} \tag{33}$$

If a thread must be continued, then the instruction associated to it ($t.run$) is executed:

$$\frac{t.susp = false \quad t.status = CONT \quad t.run, sched, env \overset{\alpha}{\longrightarrow} inst, sched', env'}{t, sched, env \overset{b}{\longrightarrow} t', sched'', env'} \tag{34}$$

where:

- $t' = t[run \ := \ inst][status \ := \ \alpha]$
- $b = false$ if $\alpha = CONT$, and $b = true$ otherwise.
- $sched'' = sched'[events \mathrel{+}= \ term(t)][move \ := \ true]$ if $\alpha = TERM$, and $sched'' = sched'$ otherwise.

Note that the *move* flag is set in case of termination to postpone the end of the current instant (otherwise, a thread waiting to join t could remain unfired).

5.2 Scheduler Execution Step

During an execution step, the scheduler gives control to all the threads that have to be continued. The threads considered are elements of the vector *actual* of the scheduler.
Nothing is done if *actual* is empty (written <>):

$$\frac{sched.actual = <>}{sched, env \overset{true}{\longrightarrow} sched, env} \tag{35}$$

If *actual* is not empty, then all elements are considered in turn (n is the length of *actual*):

$$\frac{sched_i.actual[i], sched_i, env_i \overset{b_i}{\longrightarrow} inst, sched_{i+1}[actual[i] \ := \ inst], env_{i+1} \qquad i \in 0..n-1}{sched_0, env_0 \overset{and\ b_i}{\longrightarrow} sched_n, env_n} \tag{36}$$

Note that threads are considered in a fixed order, which is the order of appearence in *actual*. The resulting boolean is true only if all the threads have finished to execute for the current instant.

5.3 Instant

Execution for an instant is finished when all threads have finished their execution for this instant:

$$\frac{sched, env \overset{true}{\longrightarrow} sched', env'}{sched, env \Longrightarrow sched', env'} \tag{37}$$

Otherwise, the scheduler cyclically performs execution steps. If no event is generated during a step (*move* is still false at the end of the step), then the scheduler decides the end of the current instant, by setting the *eoi* flag.

$$\frac{sched, env \overset{false}{\longrightarrow} sched', env' \qquad sched'', env' \Longrightarrow sched''', env''}{sched, env \Longrightarrow sched''', env''} \tag{38}$$

where $sched'' = sched'$ except that:

- $sched''.move = false$;
- $sched''.eoi = true$ if $sched'.move = false$, and $sched''.eoi = false$ otherwise.

5.4 Chaining Instants

Let *sched* be a scheduler. One defines $Next(sched)$ as the scheduler obtained from *sched* by performing the following actions in sequence:

1. concatenate *to_start* to *actual*.
2. replace *events* by *to_broadcast*.
3. remove elements of *to_stop* from *actual*; moreover, for each $t \in to\_stop$, add event $term(t)$ to *events*.
4. for each $t \in to\_resume$, set $t.susp$ to false.
5. for each $t \in to\_suspend$, set $t.susp$ to true.
6. for each thread $t \in actual$, remove it if $t.status$ is $TERM$, and otherwise set $t.status$ to $CONT$.
7. set the flag *eoi* to false.
8. reset *to_start*, *to_broadcast*, *to_stop*, *to_resume*, *to_suspend*, and *values* to empty.

The chain of instants performed by scheduler *sched*, starting from the environment *env*, is:

$$sched, env \Longrightarrow sched_1, env_1 \quad Next(sched_1), env_1 \Longrightarrow sched_2, env_2 \quad Next(sched_2), env_2 \Longrightarrow sched_3, env_3 \quad \ldots$$

Note that $Next(sched_i)$ incorporates events, threads, and orders that have been produced (by the system itself, or by the external world) during previous instant $i - 1$.

6 Conclusion

One has defined the semantics of the cooperative part of a framework for concurrent programming, based on the notion of a fair thread. Fair threads are run by fair schedulers which give threads equal rights to get the processor and equal rights to receive broadcast events.

Cooperative FairThreads has a clear and simple semantics, made of about 40 rules. The semantics is deterministic: at each step, there is no choice of the rule to apply. As a first consequence, Cooperative FairThreads is fully portable; the second consequence is that it becomes possible to reason about programs expressed in Cooperative FairThreads; finally, implementations can be very close to the semantics, which is a way to have confidence in them.

References

[1] F. Boussinot, J-F. Susini, *Java threads and SugarCubes*, Software Practice & Experience, 30(5), 545-566, 2000.

[2] F. Boussinot, *Java Fair Threads*, Inria Research Report, 2001.

[3] N. Halbwachs, *Synchronous Programming of Reactive Systems*, Kluwer Academic Pub., 1993.

[4] L. Hazard, J-F. Susini, F. Boussinot, *The Junior reactive kernel*, Inria Research Report 3732, 1999.

[5] L. Hazard, J-F. Susini, F. Boussinot, *Programming with Junior*, Inria Research Report 4027, 2000.

[6] G. Plotkin, *A Structural Approach to Operational Semantics*, Report DAIMI FN-19, Aarhus University, 1981.

[7] `http://www.inria.fr/mimosa/rp`

[8] `http://www.inria.fr/mimosa/fp/Bigloo`

[9] `http://www.inria.fr/mimosa/rp/FairThreads/FTC`.

Contents