Fairness for Dynamic Control

Jochen Hoenicke[2], Ernst-Rüdiger Olderog[1], and Andreas Podelski[2]

[1] Department für Informatik, Universität Oldenburg, 26111 Oldenburg, Germany
[2] Institut für Informatik, Universität Freiburg, 79110 Freiburg, Germany

Abstract. Already in Lamport's bakery algorithm, integers are used for fair schedulers of concurrent processes. In this paper, we present the extension of a fair scheduler from 'static control' (the number of processes is fixed) to 'dynamic control' (the number of processes changes during execution). We believe that our results shed new light on the concept of fairness in the setting of dynamic control.

1 Introduction

In Lamport's bakery algorithm [8], integers are used to express the urgency to schedule a process (the goal being to prevent the starvation of each single process by ensuring fairness). The same basic idea, though in a different realization, underlies the explicit fair scheduler of [10]. Here, the urgency to schedule a process is expressed by a possibly negative integer. The urgency increases (and the integer value decreases) if the process is enabled and not taken. The non-starvation of the process apparently relies on a lower-bound invariant: the value cannot decrease below $-n$ if n is the number of all processes. This lower bound becomes void when we move from 'static control' (the number of processes is fixed) to 'dynamic control' (the number of processes changes during execution). Indeed, the first contribution of this paper is to show that the scheduler of [10] does not ensure fairness for dynamic control; in our counterexample a process starves in an execution where it is enabled in every second step; each time when it is enabled again, its urgency has already been overtaken by some new process. This negative result opens the problem of the existence of a fair scheduler for dynamic control. We present two solutions.

The main contribution of this paper is a fair scheduler for dynamic control. The originality of this scheduler lies in a heresy. We deviate from the generally accepted believe that the non-starvation of a process relies on the well-foundedness of the corresponding sequence of integer values.

The third contribution of this paper is a different kind of fair scheduler for dynamic control. The originality of our second solution to the problem lies again in a heresy. We reformulate the problem. By dropping one of the conditions in the original definition of a fair scheduler, we arrive at a weaker notion of a scheduler (a "monitor"). The difference between a scheduler and a monitor lies in the fact that a 'monitored' execution may block. Each infinite 'monitored' execution is fair; in comparison, each 'scheduled' execution is infinite and fair. In the context of verification based on the automata-theoretic approach of Vardi-Wolper [15],

This is the author's version of the work published in *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2010)*, LNCS 6015, pages 251–265. Springer, 2010. The orginal publication is available at
www.springerlink.com/index/10.1007/978-3-642-12002-2_20

where one checks for the existence of infinite fair executions, the weaker notion of a monitor is sufficient.

In the remainder of the paper, we will present the results described above, along with a thorough investigation of a number of annexed questions. We believe that our results provide a new understanding of fairness in the context of dynamic control.

Why use explicit scheduling. There may be situations where one would like to "get rid of fairness". For example, in program analysis (whose formal foundation can be given by abstract interpretation [5]), one may want to define the semantics in terms of a (pure) transition system, i.e., a graph. Here, a popular approach is to take one of the fair schedulers used in operating systems, and to consider a new system which is composed of the original one and the scheduler, and whose semantics can be given in terms of a transition system. The objection to this approach is that the analysis result is valid only for one particular fair scheduler; i.e., it does not extend to another fair scheduler. To remove this objection, one has to take a *universal* scheduler, i.e., one that encompasses all possible fair schedulers. In contrast with schedulers implemented in operating systems, a universal scheduler is not meant to be practical. Universality holds if the scheduler is sufficiently permissive, i.e., if every possible fair execution can be scheduled (by letting the scheduler choose an appropriate sequence of alternatives at all non-deterministic choices). In order to be correct (sound), it must not be too permissive, i.e., no unfair execution can be scheduled.

Motivation of our work. Our interest for fairness in the setting of dynamic control stems from three directions.

Networked transportation systems (e.g., cars driving in groups called *platoons*) are modeled as concurrent systems (see, e.g., [2]). The fact that a traffic participant can appear and join a platoon is modeled by the creation of a new concurrent process. Fairness needs to be added as an assumption for the model for the validity of liveness properties (e.g., the termination of a merge manoeuvre between platoons).

Operating systems are typical examples of reactive systems where threads are created specifically for individual tasks. Although the execution of the overall system may be infinite, those threads must terminate in order to keep the overall system reactive. For recent automatic proof techniques addressing the termination of such threads see [13,14,12,4]. All these techniques are specifically designed to cope with fairness. Presently, however, they are restricted to the setting of static control, i.e., to the setting where the number of processes is statically fixed.

Perhaps surprisingly, recent work on model checking *safety* properties of operating systems code involve fairness [9]. Fairness is used essentially to eliminate useless (unfair) paths in the state space (i.e., paths that can be pruned without affecting the reachability of error states). This work uses explicit scheduling of the model checker for the "fair" exploration of the state space. Although the explicit scheduler in [9] is inspired by [10], it chooses a different idea for the representation of the relative urgency of processors.

Roadmap. In Section 2 we state the definitions on which we build in this paper. We formulate the classical notion of (strong) fairness not for Dijkstra's guarded command programs but, instead, for *infinitary* guarded command programs, i.e., with infinitely many branches in the **do** loops. These programs formalize the setting of *infinitary control* where infinitely many processes can be active at the same moment. In *dynamic control* only finitely many processes can be active at each moment. Then we adapt the notion of explicit scheduling and the specific scheduler for (strong) fairness from [10], which we call here $\mathbb{S}_{88}$ to the setting of infinitary control. In Section 3 we show that $\mathbb{S}_{88}$ is not valid for dynamic control. In the following we present two solutions to overcome this problem. In Section 4 we present a new scheduler $\mathbb{S}_{10}$ that is valid for dynamic control. In Section 5 we give up the requirement for a scheduler that the transition relation is total and introduce a monitor $\mathbb{M}_{88}$ derived from $\mathbb{S}_{88}$. This monitor is also valid for dynamic control. In Section 6 we investigate which of the previous results remains true in the setting of infinitary control. Section 7 concludes this paper.

2 Definitions

Though the motivation for considering fairness stems from concurrency, it is easier and more elegant to study it in terms of structured nondeterministic programs such as Dijkstra's guarded commands [7]. We follow this approach in this paper. In this section, we carry the classical definitions of fairness from Dijkstra's guarded command language over to an infinitary guarded command language, i.e., with infinitely many branches in **do** loops. It is perhaps a surprise that the definitions carry over directly. We then immediately have the definitions of fairness of programs with dynamically created processes because we will define those formally as a subclass of infinitary guarded command programs.

2.1 Dynamic Control

Our goal is a minimalistic model that allows us the study of fairness for programs with dynamically created processes. As a starting point we introduce programs with *infinitary control* by extending Dijkstra's language of guarded command programs [6] with **do** loops that have infinitely many branches. Syntactically, these **do** loops are statements of the form

$$S \equiv \mathbf{do}\ [\!]_{i=0}^{\infty}\ B_i \to S_i\ \mathbf{od} \tag{1}$$

where for each $i \in \mathbb{N}$ the *component* $B_i \to S_i$ consists of a Boolean expression B_i, its *guard*, and the statement S_i, its *command*. Therefore a component $B_i \to S_i$ is called a *guarded command* and S is called an *infinitary guarded command*.

We define the class of programs with dynamic control as a subclass of programs with infinitary control. At each moment each of the infinitely many processes "exists" (whether is has been created or not). Each process is modeled by

a branch in the infinitary **do** loop. However, at each moment, only finitely many processes have been created (or activated or allocated). All others are dormant.

Processes are referred to by natural numbers. The process (with number) i is represented by the guarded command $B_i \to S_i$. To model processes creation we use a Boolean expression cr_i for each process i such that this process is considered as being created if cr_i evaluates to true. All other processes are treated as not being created yet. It is an important assumption that a created process can disappear but not reappear, i.e., once the value of the expression cr_i has changed from true to false it cannot go back to true.

We define a structural operational semantics in the sense of Plotkin [11] for infinitary guarded commands. As usual, it is defined in terms of transitions between configurations. A *configuration* K is a pair $<S, \sigma>$ consisting a statement S that is to be executed and a state σ that assigns a value to each program variable. A *transition* is written as a step $K \to K'$ between configurations. To express termination we use the empty statement E: a configuration $<E, \sigma>$ denotes termination in the state σ. For a Boolean expression B we write $\sigma \models B$ if B evaluates to true in the state σ. Process i is *created* in a state σ if $\sigma \models cr_i$ and it is *enabled* in state σ if it is created and its guard B_i evaluates to true, formally, $\sigma \models cr_i \wedge B_i$.

For the infinitary **do** loop S as in (1) we have two cases of transitions:

1. $<S, \sigma> \to <S_i;S, \sigma>$ if $\sigma \models cr_i \wedge B_i$ for each $i \in \mathbb{N}$,
2. $<S, \sigma> \to <E, \sigma>$ if $\sigma \models \bigwedge_{i=1}^{\infty} \neg(cr_i \wedge B_i)$.

Case 1 states that each *enabled* component $B_i \to S_i$ of S, i.e., with both the expression cr_i and the guard B_i evaluating to true in the current state σ, can be entered. If more than one component of S is enabled, one of them will be chosen nondeterministically. The successor configuration $<S_i;S, \sigma>$ formalizes the repetition of the **do** loop: once the command S_i is executed the whole loop S has to be executed again. Formally, the transitions of the configuration $<S_i;S, \sigma>$ are determined by the transition rules for the other statements of the guarded command language. For further details see, e.g., [1]. Case 2 states that the **do** loop terminates if none of the components is enabled any more, i.e, if all expressions $cr_i \wedge B_i$ evaluate to false in the state σ.

In this paper we investigate programs with only *one* infinitary **do** loop S of the form (1). This simplifies its definition of fairness and is sufficient for modeling dynamic control. An *execution* of S starting in a state σ_0 is a sequence of transitions

$$K_0 \to K_1 \to K_2 \to \dots, \tag{2}$$

with $K_0 = <S, \sigma_0>$ as the initial configuration, which is either infinite or maximally finite, i.e., the sequence cannot be extended further by some transition.

Consider a program S of the form (1). Then for S having *infinitary control* there is no further requirement on the set of created processes. A program S has *dynamic control* if for every execution (2) of S the set of created processes is finite in every state of a configuration in (2).

A program S has *bounded control* if for every execution (2) there exists some $n \in \mathbb{N}$ such that the number of created processes is bounded by n in every state of a configuration in (2). A program S has *static control* if there is a fixed finite set F of processes such that for every execution (2) the set of created processes is contained in F in every state of a configuration in (2).

Note that we have the following hierarchy: programs with static control are a special case of programs with bounded control, which are a special case of programs with dynamic control, which in turn are a special case of programs with infinitary control.

2.2 Fairness

In this paper we extend the definition of fairness[3] of [10] from programs with static control to programs with process creation and infinitary control. Since fairness can be expressed in terms of created, enabled, and selected processes only, we abstract from all other details in executions and define it on runs.

We now fix an execution as in (2) and define the corresponding run. A transition $K_j \to K_{j+1}$ with $j \in \mathbb{N}$ is a *select transition* if it consists of the selection of an enabled process of S, formally, if $K_j = <S, \sigma>$ and $K_{j+1} = <S_i;S, \sigma>$ with $\sigma \models cr_i \wedge B_i$ for some $i \in \mathbb{N}$, so process i has been *selected* for execution in this transition. We define the *selection* of the transition $K_j \to K_{j+1}$ as the triple (C_j, E_j, i_j), where C_j is the set of all created processes, i.e.,

$$C_j = \{i \in \mathbb{N} \mid \sigma \models cr_i\},$$

and E_j is the subset of all enabled processes, i.e.,

$$E_j = \{i \in C_j \mid \sigma \models B_i\},$$

and i_j is the (index of the) selected process, i.e., $i_j = i$. Obviously, the selected command is among the enabled components. A *run of the execution* (2) is the sequence of all its selections, formally, the sequence

$$(C_{j_0}, E_{j_0}, i_{j_0})(C_{j_1}, E_{j_1}, i_{j_1}) \ldots$$

such that $C_{j_0} C_{j_1} \ldots$ is the subsequence of configurations with outgoing select transitions. Computations that do not pass through any select transition yield the empty run. A *run of a program* S is the run of one of its executions.

A run

$$(C_0, E_0, i_0)(C_1, E_1, i_1)(C_2, E_2, i_2) \ldots \tag{3}$$

is called *fair* if it satisfies the condition

$$\forall i \in \mathbb{N} : (\overset{\infty}{\exists} j \in \mathbb{N} : i \in E_j \to \overset{\infty}{\exists} j \in \mathbb{N} : i = i_j).$$

[3] In the literature, this notion of fairness is qualified as *strong fairness* (or *compassion*). For brevity, we simply refer to this notion without the qualifier in this paper.

where the quantifier $\overset{\infty}{\exists}$ denotes "there exist infinitely many". By our assumption (see Subsection 2.1), the fact that the process i is infinitely often enabled, formally $\overset{\infty}{\exists} j \in \mathbb{N} : i \in E_j$, implies by $E_j \subseteq C_j$ that process i is created at some moment and stays created forever, formally $\exists j_0 \in \mathbb{N}\ \forall j \geq j_0 : i \in C_j$.

In a fair run, every process i which is enabled infinitely often, is selected infinitely often. Note that every finite run is trivially fair. An *execution* of a program S of the form (1) is *fair* if its run is fair. Thus for fairness only select transitions are relevant; transitions inside the commands S_i of S do not matter. Again, every finite execution is trivially fair. Thus we concentrate on infinite executions throughout this paper.

Although we are not interested in the case where infinitely many processes can be enabled at the same time (continuously or infinitely often) and although this case is perhaps not practically relevant, the definition of fairness still makes sense, i.e., there exist fair executions in this case.

2.3 Explicit Scheduling

We extend the definition of a scheduler from [10] to the setting of infinitary control. In a given state σ the scheduler inputs a set C of created processes and a subset $E \subseteq C$ of enabled processes. It outputs some process $i \in E$ and transitions to a new state σ'. We require that the scheduler is totally defined, i.e., for every scheduler state and every input set E the scheduler will produce an output $i \in E$ and update its scheduler state. Thus a scheduler can never block the execution of a program but only influence its direction. Summarizing, we arrive at the following definition.

Definition 1 ([10]). *A* scheduler *is a triple* $\mathbb{S} = (\Sigma, \Sigma_0, \delta)$, *where*

- Σ *is a set of* states *with typical element* σ,
- $\Sigma_0 \subseteq \Sigma$ *is the set of* initial *states, and*
- δ *is a* transition relation *of the form*

$$\delta \subseteq \Sigma \times 2^{\mathbb{N}} \times 2^{\mathbb{N}} \times \mathbb{N} \times \Sigma$$

which is total *in the following sense:*

$$\forall \sigma \in \Sigma\ \forall C \in 2^{\mathbb{N}}\ \forall E \in 2^C \setminus \{\emptyset\}\ \exists i \in E\ \exists \sigma' \in \Sigma : (\sigma, C, E, i, \sigma') \in \delta.$$

Thus for every state σ, *every set* C *of created processes, and every nonempty subset* $E \subseteq C$ *of enabled processes there exists a process* $i \in E$ *and an the updated state* σ' *such that the tuple* $(\sigma, C, E, i, \sigma')$ *satisfies the transition relation* δ.

A run $(C_0, E_0, i_0)(C_1, E_1, i_1)(C_2, E_2, i_2)\ldots$ *is* produced *by a scheduler* $\mathbb{S}$ *if there exists an infinite sequence* $\sigma_0\sigma_1\sigma_2\ldots \in \Sigma^\omega$ *with* $\sigma_0 \in \Sigma_0$ *such that*

$$(\sigma_j, C_j, E_j, i_j, \sigma_{j+1}) \in \delta$$

holds for all $j \in \mathbb{N}$. *A scheduler* $\mathbb{S}$ *is* sound *if every run that is produced by* $\mathbb{S}$ *is fair. A scheduler* $\mathbb{S}$ *is* universal *if every fair run is produced by* $\mathbb{S}$. *A scheduler* $\mathbb{S}$ *is* valid *if it is both sound and universal.*

2.4 The scheduler $\mathbb{S}_{88}$

The explicit schedulers given in [10] use auxiliary integer-valued variables (so-called *scheduling variables*), one for each process, to keep track of the relative urgency of each process (relative to the other processes). Making it more urgent is implemented by decrementing its scheduling value. Thus, scheduling values can become negative. The crucial step is the non-deterministic update to a *non-negative* integer each time after the process has been selected. Then, the process is not necessarily less urgent than all other processes. However, it is definitely less urgent than those that already have a negative scheduling value. This fact is used to prove (by induction) the *scheduling invariant*: the scheduling value will never decrease below $-n$, where n is the number of all processes [10]. This again means that a process cannot become "arbitrarily urgent"; i.e., it has to be selected after it has been made more urgent a finite (though unboundedly large) number of times, which is exactly what fairness means.

In [10] a scheduler for fairness of programs with static control was proposed. We extend it here to the case of infinitely many components and call it $\mathbb{S}_{88}$. With each process i it associates a *scheduling variable* $z[i]$ representing a priority assigned to that process. A process i has a higher priority than a process j if $z[i] < z[j]$ holds.

Definition 2 ([10]). *The scheduler* $\mathbb{S}_{88} = (\Sigma, \Sigma_0, \delta)$ *is defined as follows:*

- *The states* $\sigma \in \Sigma$ *are given by the values of an infinitary array* z *of type* $\mathbb{N} \to \mathbb{Z}$*, i.e.,* $z[i]$ *is a positive or negative integer for each* $i \in \mathbb{N}$*.*
- *The initial states in* Σ_0 *are those where each scheduler variable* $z[i]$ *has some nonnegative integer value.*
- *The relation* $(\sigma, C, E, i, \sigma') \in \delta$ *holds for states* $\sigma, \sigma' \in \Sigma$*, a set* C *of created processes, a set* $E \subseteq C$ *of enabled processes, and a process* $i \in E$ *if the value of* $z[i]$ *is minimal in* σ*, i.e., if*

$$z[i] = min\{z[k] \mid k \in E\}$$

holds in σ*, and* σ' *is obtained from* σ *by executing the following statement:*

$$\begin{aligned} \mathit{UPDATE}_i \equiv\ & z[i] := ?; \\ & \textbf{for all}\ \ j \in E \setminus \{i\}\ \textbf{do}\ z[j] := z[j] - 1\ \textbf{od}. \end{aligned}$$

Note that the transition relation δ is total as required by Definition 1. The update of the scheduling variables guarantees that the priorities of all enabled but not selected processes j are increased. The priority of the selected process i, however, is reset arbitrarily. The idea is that by gradually increasing the priority of enabled processes, their activation cannot be refused forever.

3 The scheduler $\mathbb{S}_{88}$ and dynamic control

For static control the scheduler $\mathbb{S}_{88}$ is valid, i.e., sound and universal as shown in [10]. A closer examination of the proof shows that this result extends to bounded control. However, for dynamic control this does not hold any more.

Theorem 1. *The scheduler $\mathbb{S}_{88}$ is not valid for dynamic control.*

Proof. We show that $\mathbb{S}_{88}$ is *not sound* for programs with dynamic control. To this end, we construct a run produced by $\mathbb{S}_{88}$ in which process 0 is treated unfair, i.e., it is infinitely often enabled but never selected. The idea is that in each step a new process is created, which is enabled all the time. The process 0 is only enabled in every second step. The scheduling variable of the other processes will decrease more rapidly than the scheduling variable of process 0 and thus will overtake it. The values of scheduling variables will force $\mathbb{S}_{88}$ to activate the newly created processes rather than process 0.

i	σ_0	σ_1	σ_2	σ_3	σ_4	σ_5	σ_6	σ_7	σ_8	σ_9	...
0	(0)	0	(-1)	-1	(-2)	-2	(-3)	-3	(-4)	-4	...
1	0*	0	-1*	0	-1	-2	-3*	0	-1	-2	...
2	0	-1*	0	-1	-2*	0	-1	-2	-3	-4	...
3		0	-1	-2*	0	-1	-2	-3	-4*	0	...
4			0	-1	-2	-3*	0	-1	-2	-3	...
5				0	-1	-2	-3	-4*	0	-1	...
⋮					⋱						...

Table 1. A run where process 0 is treated unfair

Table 1 shows an initial segment of this run in detail. In the column denoted by i the process numbers are shown. The other columns show the values of the scheduling variables $z[i]$ in the scheduler states $\sigma_0, \sigma_1, \sigma_2, \ldots$. A star $*$ after a value indicates that in this state the process in the corresponding row is selected. For example, in state σ_0 process 1 is selected. An entry in parenthesis indicates that in this state the corresponding process is not enabled. This is the case only for process 0. If process 0 is not enabled its scheduling variable $z[0]$ is not decremented in the next step. Empty boxes in the table indicate that in this state the corresponding process is *not yet created*, otherwise the process is created. Thus in state σ_0 only the processes 0, 1, and 2 are created. Note that in each step a newly created process appears in the successor state.

In general, in each state σ_{2n} process 0 is not enabled, its priority is $-n$, and for each $z \in \{-n, \ldots, 0\}$ there are exactly two processes different from process 0 with priority z. In state each σ_{2n+1} process 0 is enabled, its priority is still $-n$, there is one process with priority $-n-1$, and for each $z \in \{-n, \ldots, 0\}$ there are again exactly two processes different from process 0 with priority z: the process scheduled in the previous step and the new process have priority 0 and the two processes with priority $z+1$ in the previous step have now priority z. Then the single process with priority $-n-1$ is scheduled and we arrive at state σ_{2n+2}, where process 0 has priority $-n-1$ and there are two processes different from 0 for each priority $z \in \{-n-1, \ldots, 0\}$. This concludes the proof. □

It is interesting to notice the following.

Remark 1. The scheduler $\mathbb{S}_{88}$ is universal for dynamic control.
The proof idea is that at each moment the value of the scheduling variable $z[i]$ of process i is set to the number of times process i is enabled before i is selected or disappears or gets disabled forever. In this construction the variables $z[i]$ have at each moment nonnegative values. The selected process i has the scheduling value $z[i] = 0$. All other enabled processes j have scheduling values $z[j] \geq 1$.

An alternative scheduler for fairness was proposed in [3], Chapter 6. There it is shown that this scheduler is, in our terminology, valid for static control. However, by a variant of the counterexample in Table 1 it can be shown that also this scheduler is unsound for dynamic control.

4 The scheduler $\mathbb{S}_{10}$

We obtain the scheduler $\mathbb{S}_{10}$ from $\mathbb{S}_{88}$ by the applying the decrement of the scheduling variable to all created processes $j \in C \setminus \{i\}$ and not only to the enabled processes $j \in E \setminus \{i\}$.

Definition 3. *The scheduler* $\mathbb{S}_{10}$ *results from* $\mathbb{S}_{88}$ *by replacing* $UPDATE_i$ *with*

$$\begin{aligned} S\text{-}UPDATE_i \equiv\ & z[i] := ?; \\ & \textbf{for all}\ \ j \in C \setminus \{i\}\ \textbf{do}\ z[j] := z[j] - 1\ \textbf{od}. \end{aligned}$$

Theorem 2. *The scheduler* $\mathbb{S}_{10}$ *is valid for dynamic control.*

Proof. We show that $\mathbb{S}_{10}$ is both sound and universal for dynamic control.
Soundness. Consider a run

$$(C_0, E_0, i_0)\ldots(C_j, E_j, i_j)\ldots \tag{4}$$

of a program of the form (1) with dynamic control that is produced by $\mathbb{S}_{10}$ using the sequence $\sigma_0\,\sigma_1\ldots\sigma_j\,\sigma_{j+1}\ldots$ of scheduler states. We claim that (4) is fair.

Suppose the contrary holds. Then there exists some process i that is enabled infinitely often, but from some moment on never selected. Formally, for some $j_0 \geq 0$

$$(\overset{\infty}{\exists} j \in \mathbb{N} : i \in E_j) \wedge (\forall j \geq j_0 : i \neq i_j)$$

holds in (4). Then the variable $z[i]$ of $\mathbb{S}_{10}$, which gets decremented whenever the process i is not selected, becomes arbitrarily small. Thus we can choose j_0 large enough so that $z[i] < 0$ holds in σ_{j_0}. Consider the set

$$Cr_{i,j} = \{k \in \mathbb{N} \mid k \in C_j \wedge \sigma_j \models z[k] \leq z[i]\}$$

of all created processes in C_j whose priority is least that of the neglected process i, formally, whose scheduling variable has at most the value of the scheduling variable of i. Since we consider dynamic control, Cr_{i,j_0} is finite in σ_{j_0}.

We show that for all $j \geq j_0$:

$$Cr_{i,j+1} \subseteq Cr_{i,j} \quad \text{and} \quad Cr_{i,j+1} \neq Cr_{i,j} \text{ if } i \in E_j. \tag{5}$$

Consider a process p that was not in $Cr_{i,j}$. We show $p \notin Cr_{i,j+1}$ to prove the inclusion. If p was scheduled in step j, then $\sigma_{j+1} \models z[i] < 0 \leq z[p]$, thus $p \notin Cr_{i,j+1}$.

If process p is newly created in step j we exploit two facts. (1) By the definition of *S-UPDATE*$_i$, its scheduling variable $z[p]$ is not decremented as long as p is not created. (2) The process p has not been created before by the assumption that a created process can disappear but not reappear, stated in Subsection 2.1. By (1) and (2), $z[p]$ has still its initial nonnegative value in state σ_{j+1}, thus $\sigma_{j+1} \models z[p] \geq 0$. So $p \notin Cr_{i,j+1}$.

If we take a process p different from the selected process then in the successor state σ_{j+1} the validity of the inequality $z[p] \leq z[i]$ is preserved (both p and i have their scheduling variable decremented by the definition of *S-UPDATE*$_i$).

If process i is enabled in step j, the scheduler needs to select a process p from $Cr_{i,j}$. As seen before, the scheduled process is not in $Cr_{i,j+1}$, thus $Cr_{i,j} \neq Cr_{i,j+1}$. This proves property (5).

By assumption i is enabled infinitely often, so by (5) the set $Cr_{i,j}$ is strictly decreasing infinitely often. This contradicts the fact that Cr_{i,j_0} is finite.

Universality. Consider a fair run

$$(C_0, E_0, i_0)(C_1, E_1, i_1)(C_2, E_2, i_2)\ldots. \tag{6}$$

We show that (6) can be produced by $\mathbb{S}_{10}$ by constructing a sequence $\sigma_0 \ldots \sigma_j \ldots$ of scheduler states satisfying $(\sigma_j, C_j, E_j, i_j, \sigma_{j+1}) \in \delta$ for every $j \in \mathbb{N}$. The construction proceeds by assigning appropriate values to the scheduling variables $z[i]$ of $\mathbb{S}_{10}$. For $i, j \in \mathbb{N}$ we put

$$\sigma_j(z[i]) = |\{k \in \mathbb{N} \mid j \leq k < m_{i,j} \wedge i \in C_k\}| - |\{k \in \mathbb{N} \mid m_{i,j} \leq k < j \wedge i \in C_k\}|\,,$$

where

$$m_{i,j} = min \left\{ m \in \mathbb{N} \;\middle|\; \begin{array}{l} (1)\ (j \leq m \wedge i_m = i) \\ \quad \vee \\ (2)\ (\forall n \geq m : i \notin E_n) \end{array} \right\}.$$

Note that $m_{i,j}$ is the minimum of a non-empty subset of $\mathbb{N}$ because the run (6) is fair. In case (1) of the definition of $m_{i,j}$, i.e., when i is eventually selected, the value $\sigma_j(z[i])$ is nonnegative. However, in case (2) of the definition of $m_{i,j}$, i.e., when i is not enabled any more, the value $\sigma_j(z[i])$ can denote arbitrarily negative values.

This construction of values $\sigma_j(z[i])$ is possible with the assignments in $\mathbb{S}_{10}$. In the constructed run the selected process i has the scheduling value $z[i] = 0$. All other enabled processes j have scheduling values $z[j] \geq 1$. So i is the unique enabled process with the minimum of all scheduling values, which is 0. □

5 The monitor $\mathbb{M}_{88}$

The scheduler $\mathbb{S}_{88}$ does not decrease the scheduling variables of processes that are not enabled. So these scheduling variables cannot become arbitrarily negative. However, $\mathbb{S}_{88}$ it not valid for dynamic control. The new scheduler $\mathbb{S}_{10}$ is valid for dynamic control but the scheduling variables can become arbitrarly negative for created processes that are from some moment on never enabled any more.

In this section we shall propose a variant of $\mathbb{S}_{88}$ where the scheduling variables are prevented from becoming negative. The price we pay for this property is that this may lead to a blocking behaviour. Schedulers are required to be nonblocking, i.e., they should have a totally defined transition relation. We now drop this requirement and call the resulting device a *monitor*.

Definition 4. *A* monitor *is a triple* $\mathbb{M} = (\Sigma, \Sigma_0, \delta)$*, where*

- Σ *is a set of* states *with typical element* σ*,*
- $\Sigma_0 \subseteq \Sigma$ *is the set of* initial *states, and*
- δ *is a* transition relation *of the form*

$$\delta \subseteq \Sigma \times 2^{\mathbb{N}} \times 2^{\mathbb{N}} \times \mathbb{N} \times \Sigma$$

(without totality requirement as for schedulers).

A run $(C_0, E_0, i_0)(C_1, E_1, i_1)(C_2, E_2, i_2)\ldots$ *is* accepted *by a monitor* $\mathbb{M}$ *if there exists an infinite sequence* $\sigma_0\sigma_1\sigma_2\ldots \in \Sigma^\omega$ *with* $\sigma_0 \in \Sigma_0$ *such that*

$$(\sigma_j, C_j, E_j, i_j, \sigma_{j+1}) \in \delta$$

holds for all $j \in \mathbb{N}$*. A monitor* $\mathbb{M}$ *is* sound *if every run that is accepted by* $\mathbb{M}$ *is fair. A monitor* M *is* universal *if every fair run is accepted by* $\mathbb{M}$*. A monitor* $\mathbb{M}$ *is* valid *if it is both sound and universal.*

Since the totality requirement is dropped for the transition relation δ, the monitor cannot be used to produce a fair run step-by-step because for a given scheduler state σ, a set C of created processes, and a set E of enabled processes there may not be a process $i \in E$ and an updated scheduler state σ' with $(\sigma, C, E, i, \sigma') \in \delta$. However, a monitor can be used as an acceptor of given runs. Then the question of being able to stepwise produce the run is not relevant.

We modify the scheduler $\mathbb{S}_{88}$ of Definition 2 to a monitor called $\mathbb{M}_{88}$.

Definition 5. *The monitor* $\mathbb{M}_{88}$ *is obtained from the scheduler* $\mathbb{S}_{88}$ *by changing the type of the infinitary array* z *of scheduling variables to* $\mathbb{N} \to \mathbb{N}$*, i.e., for each process* $i \in \mathbb{N}$ *the scheduling variable* $z[i]$ *can store only* nonnegative *integers. As a consequence, inside the statement* $UPDATE_i$ *each decrement operation*

$$z[j] := z[j] - 1$$

is defined only if $z[j] > 0$ *holds. Otherwise the operation will cause a failure, which blocks any further execution.*

As in the scheduler $\mathbb{S}_{88}$ the process i with the minimal value of the scheduling variables among the enabled processes is selected. However, in contrast to $\mathbb{S}_{88}$ and $\mathbb{S}_{10}$ the transition relation of the monitor $\mathbb{M}_{88}$ is not totally defined any more. Nevertheless, we have the following result.

Theorem 3. *The monitor $\mathbb{M}_{88}$ is valid for dynamic control.*

Proof. We show that $\mathbb{M}_{88}$ is both sound and universal for dynamic control.

Soundness. Consider a run

$$(C_0, E_0, i_0)\ldots(C_j, E_j, i_j)\ldots \tag{7}$$

of a program of the form (1) with dynamic control that is accepted by $\mathbb{M}_{88}$, and let $\sigma_0\ldots\sigma_j\ldots$ be a sequence of states with $(\sigma_j, C_j, E_j, i_j, \sigma_{j+1}) \in \delta$ for every $j \in \mathbb{N}$. We claim that (7) is fair.

Suppose the contrary holds. Then there exists some process i which is infinitely often enabled, but from some moment on never selected. Note that whenever process i is enabled but not selected, the monitor $\mathbb{M}_{88}$ decrements its scheduling variable $z[i]$ *provided* $z[i] > 0$ holds. However, $z[i]$ cannot be decremented infinitely often without raising a failure, *Contradiction.*

Universality. Let the (7) be fair. Then we can proceed as in the proof outlined for Remark 1 because according to that construction in each step of the run exactly the selected process i has the scheduling value $z[i] = 0$. All other enabled processes have scheduling values $z[j] \geq 1$. Thus the monitor $\mathbb{M}_{88}$ can simulate the scheduler $\mathbb{S}_{88}$. □

The scheduling variables of the scheduler $\mathbb{S}_{88}$ when applied to programs with n processes (static control) can become arbitrarily positive but *not arbitrarily negative*, i.e., they do not assume values below $-n$ due to an execution invariant of $\mathbb{S}_{88}$(see [10]). By contrast, the scheduling variables of $\mathbb{S}_{10}$ *can become arbitrarily negative* even when it is applied to programs with static comtrol only. By definition, the scheduling variables of the monitor $\mathbb{M}_{88}$ stay nonnegative. The price for this is that the monitor *can block* the computation.

Other monitors

The monitor $\mathbb{M}_{88}$ selects a process i with the minimal value of the scheduling variables among the enabled processes. We discuss two variants of this choice. Let $\mathbb{M}_{88}{}^{*}$ result from $\mathbb{M}_{88}$ by selecting an enabled process i with $z[i] = 0$, and $\mathbb{M}_{88}{}^{**}$ result from $\mathbb{M}_{88}$ by selecting an *arbitrary* enabled process.

Remark 2. The monitors $\mathbb{M}_{88}{}^{*}$ and $\mathbb{M}_{88}{}^{**}$ are valid for dynamic control.

Proof. A closer inspection of the proof of Theorem 3 shows that the soundness argument is independent of how an enabled process is selected. For the universality argument we notice that in the construction of the monitor state sequence always an enabled process i with $z[i] = 0$ is selected. □

Surprisingly, an attempt to modify the scheduler $\mathbb{S}_{10}$ to a corresponding monitor $\mathbb{M}_{10}$ fails because we can show that this monitor is not valid. Indeed, let us define the monitor $\mathbb{M}_{10}$ analogously to Definition 5 by changing in the scheduler $\mathbb{S}_{10}$ the type of the infinitary array z of scheduling variables to $\mathbb{N} \to \mathbb{N}$, i.e., for each process $i \in \mathbb{N}$ the scheduling variable $z[i]$ can store *only nonnegative* integers. Again, the transition relation of the monitor $\mathbb{M}_{10}$ is not totally defined because the decrement operations $z[j] := z[j] - 1$ can fail.

In contrast to the monitor $\mathbb{M}_{88}$, we have the following negative result.

Remark 3. The monitor $\mathbb{M}_{10}$ is not valid for dynamic control, not even for static control.

Proof. We show that $\mathbb{M}_{10}$ is *not universal* for programs with *static control*. Consider a fair run of a program where from some moment on a created process j is not enabled any more. Then the corresponding variable $z[j]$ gets decremented whenever another process i is selected. So $z[j] = 0$ will eventually hold and thus the run cannot be accepted by $\mathbb{M}_{10}$ without blocking. □

6 Infinitary fairness

In this section we investigate which of our previous results actually relies on the restriction to dynamic control. We shall see that some hold even in the setting of infinitary control and others do not.

Since the scheduler $\mathbb{S}_{88}$ is not valid for dynamic control, it is not valid for infinitary control either. More precisely, $\mathbb{S}_{88}$ is not sound for infinitary control. This follows trivially from the corresponding argument in the proof of Theorem 1 for dynamic control. On the other hand, $\mathbb{S}_{88}$ is universal for infinitary control. Indeed, the proof idea presented for Remark 1 does not rely on the restriction to dynamic control.

For the scheduler $\mathbb{S}_{10}$ we have analogous results for infinitary control.

Theorem 4. *The scheduler $\mathbb{S}_{10}$ is not valid for infinitary control.*

Proof. The soundness argument in the proof of Theorem 2 exploits the assumption of dynamic control. We show now that $\mathbb{S}_{10}$ is *not sound* for programs with infinitary control. To this end, we construct a run produced by $\mathbb{S}_{10}$ where *every* process is treated unfair. More precisely, every process is always enabled but selected *only once*, in the ith selection of the run: $(\mathbb{N},\mathbb{N},0)(\mathbb{N},\mathbb{N},1)(\mathbb{N},\mathbb{N},2)\ldots$ This is possible by choosing the corresponding sequence $\sigma_0\sigma_1\sigma_3\ldots$ of scheduler states as follows:

i	σ_0	σ_1	σ_2	σ_3	σ_4	$\ldots$
0	0*	0	-1	-2	-3	$\ldots$
1	0	-1*	0	-1	-2	$\ldots$
2	0	-1	-2*	0	-1	$\ldots$
3	0	-1	-2	-3*	0	$\ldots$
4	0	-1	-2	-3	-4*	$\ldots$
$\ldots$	$\ldots$	$\ldots$	$\ldots$	$\ldots$	$\ldots$	$\ldots$

$$\sigma_j(z[i]) = \begin{cases} i+1-j & \text{if } i < j \\ -j & \text{if } i \geq j \end{cases}$$

The table on the previous page shows an initial segment of this sequence in detail. As in Table 1, in the column denoted by i the process numbers are shown. The other columns in the table show the values of the scheduling variables $z[i]$ in the scheduler states $\sigma_0, \sigma_1, \sigma_2, \ldots$. A star $*$ after a value indicates that in this state the process in the corresponding row is selected. For example, in state σ_0 process 0 is selected. □

On the other hand, the universality of $\mathbb{S}_{10}$ still holds for infinitary control. Indeed, the universality argument in the proof of Theorem 2 does not use the assumption of dynamic control. What about the monitor $\mathbb{M}_{88}$? Interestingly, it can also be used for infinitary control.

Remark 4. The monitor $\mathbb{M}_{88}$ is valid for infinitary control.
This result follows from a closer examination of the proof of Theorem 3 which does not use the assumption of dynamic control.

7 Conclusion

The results presented in this paper provide a new understanding of fairness in the context of dynamic control. Fairness means the non-starvation of each single process. I.e, it must not happen that a process, say i, is enabled infinitely often but not taken. Thus, at each state σ of an execution, there is only a finite number of positions where process i is enabled before it is taken. The difficulty of scheduling a dynamically growing number of processes stems from the need to prioritize "fairly" among the processes.

Our first result says that correlating the priority to the number of times that a process was enabled but not taken does not guarantee fairness. Since more and more newly created processes can increase their priority, it is possible that one of them overtakes process i in its priority.

Our second result says that correlating the priority to the number of times that a process was not taken (regardless of whether it was enabled or not) prevents this kind of overtaking and succeeds in guaranteeing fairness. As a consequence, the priority of a process that is never enabled again can get arbitrarily high, and in particular higher than the priority of every enabled process. This fact, although it contradicts the original intuition about explicit scheduling, does not impede the functioning of the scheduler.

The third result says that correlating the priority to the number of times that a process was enabled but not taken does guarantee fairness *if* this can happen only a finite number of times. Which itself is enforced by a bound on its priority; i.e., the priority cannot increase indefinitely. As a result, one obtains blocked executions (the execution gets blocked if the bound is reached). Although one needs to overcome a conceptual barrier (since blocking contradicts the philosophy that underlies the very concept of scheduling), one arrives at the concept of a monitor which fullfills the purpose of a scheduler in the context of verification.

This leads to a new line of future research: explore the potential of the monitor for automated verification methods for termination and liveness properties. Tools for programs that have terminating, though unboundedly long executions, in general use integer arithmetic to deal with ranking functions. The concept of the integer variable that measures the priority of a process is related to the concept of a rank and requires the same kind of reasoning; i.e., adding an integer-based monitor to deal with fairness does not add a foreign element as far as the reasoning method is concerned. For this reason, the potential of the monitor for automated verification methods seems promising.

We have left open the following question. Does there exists an explicit scheduler for infinitary control?

Acknowledgements. This work was partly supported by the German Research Council (DFG) as part of the Transregional Collaborative Research Center "Automatic Verification and Analysis of Complex Systems" (SFB/TR 14 AVACS). We thank Andrey Rybalchenko for helpful comments on this paper.

References

1. K.-R. Apt and E.-R. Olderog. *Verification of Sequential and Concurrent Programs.* Springer, 2nd edition, 1997.
2. J. Bauer, I. Schaefer, T. Toben, and B. Westphal. Specification and verification of dynamic communication systems. In K. Goossens and L. Petrucci, editors, *ACSD*, Turku, Finland, 2006. IEEE.
3. E. Best. *Semantics of Sequential and Parallel Programs.* Prentice Hall, 1996.
4. B. Cook, A. Podelski, and A. Rybalchenko. Proving thread termination. In *PLDI*. ACM Press, 2007.
5. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixedpoints. In *POPL*, pages 238–252. ACM, 1977.
6. E. W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Comm. of the ACM*, 18:453–457, 1975.
7. N. Francez. *Fairness.* Springer-Verlag, New York, 1986.
8. L. Lamport. A new solution of Dijkstra's concurrent programming problem. *Comm. of the ACM*, 17(8):453–455, 1974.
9. M. Musuvathi and S. Quadeer. Fair stateless model checking. In *PLDI*, June 2008.
10. E. R. Olderog and K. R. Apt. Fairness in parallel programs, the transformational approach. *ACM TOPLAS*, 10:420–455, 1988.
11. G. Plotkin. A structural approach to operational semantics. *J. of Logic and Algebraic Programming*, 60–61:17–139, 2004.
12. A. Pnueli, A. Podelski, and A. Rybalchenko. Separating fairness and well-foundedness for the analysis of fair discrete systems. In N. Halbwachs and L. Zuck, editors, *TACAS*, volume 3440 of *LNCS*, pages 124–139. Springer, 2005.
13. A. Podelski and A. Rybalchenko. Transition invariants. In *LICS '04*, pages 32–41. IEEE Computer Society, 2004.
14. A. Podelski and A. Rybalchenko. Transition predicate abstraction and fair termination. In *POPL*, pages 132–144. ACM, 2005.
15. M. Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *LICS*, pages 332–344. IEEE Computer Society, 1986.