CCS: It’s not Fair!
Fair Schedulers cannot be implemented in CCS-like languages even under progress and certain fairness assumptions

Rob van Glabbeek · Peter Höfner

Abstract In the process algebra community it is sometimes suggested that, on some level of abstraction, any distributed system can be modelled in standard process-algebraic specification formalisms like CCS. This sentiment is strengthened by results testifying that CCS, like many similar formalisms, is Turing powerful and provides a mechanism for interaction. This paper counters that sentiment by presenting a simple fair scheduler—one that in suitable variations occurs in many distributed systems—of which no implementation can be expressed in CCS, unless CCS is enriched with a fairness assumption.

Since Dekker’s and Peterson’s mutual exclusion protocols implement fair schedulers, it follows that these protocols cannot be rendered correctly in CCS without imposing a fairness assumption. Peterson expressed this algorithm correctly in pseudocode without resorting to a fairness assumption, so it furthermore follows that CCS lacks the expressive power to accurately capture such pseudocode.

NICTA is funded by the Australian Government through the Department of Communications and the Australian Research Council through the ICT Centre of Excellence Program.

R. J. van Glabbeek
NICTA and UNSW. E-mail: rvg@cs.stanford.edu

P. Höfner
NICTA and UNSW. E-mail: Peter.Hoefner@nicta.com.au
Part I Motivation & Discussion

1 Background

In the process algebra community it is generally taken for granted that, on some level of abstraction, any distributed system can be modelled in standard process-algebraic specification formalisms like CCS [45].

Of course, if a distributed system has features related to time, probability, broadcast communication or anything else that is not innately modelled in CCS, yet essential to adequately describe the distributed system under consideration, appropriate extensions are needed, such as timed process algebras (e.g., [56, 35, 3, 43, 16]), probabilistic process algebras (e.g., [34]) or calculi for broadcast communication (e.g., [54]). This paper is not concerned with such features.

The relative expressiveness of process algebras is a well-studied subject [58, 50, 32], and in this area CCS-like process algebras are considered far from universally expressive. In [27] for instance it is pointed out that the parallel composition operator of CSP [11, 36] cannot be expressed in CCS. The priority operator of [5] is a good example of an operator that cannot be expressed in any of the standard process algebras such as CCS, CSP, ACP [7] or LOTOS [9]. These results focus, however, on the possibility of expressing operators—composing a process out of one or more components—as CCS-contexts; they cast no doubt on the possibility of expressing actual processes in CCS.

Beside operators, it has also be shown that there exist examples of process specifications that cannot be faithfully rendered in CCS-like formalisms (cf. [26]). We will illustrate this in Section 3. In this paper we distinguish process specifications from actual processes that could in principle be implemented and executed. Again, the evidence presented casts no doubt on the possibility of expressing actual processes in CCS.

Incorporating these clarifications of our meaning, we expect that many concurrency experts feel that, up to an adequate level of abstraction, any reactive system can be rendered in CCS. This sentiment is strengthened by results testifying that CCS, like many similar formalisms, is Turing powerful [45]. As a manifestation of this, any computable partial function \( f : \Sigma^* \rightarrow \Sigma^* \) over some finite alphabet \( \Sigma \) can be modelled by a CCS context \( P[\omega] \), such that, for any input word \( w = a_1a_2\ldots a_n \in \Sigma^* \), encoded as a CCS expression \( W := a_1a_2\ldots a_n.z.0 \) featuring an end-of-input marker \( z \), the process \( P[W] \) computes forever without performing any visible actions if \( f(w) \) is undefined, and otherwise performs the sequence of visible actions \( f(w) \), followed by an end-of-output marker \( z \).

It is sometimes argued that Turing machines are an inadequate formalism to capture interactive behaviour as displayed by today’s computers [62, 11]. The main argument is that Turing machines are function-based and calculate, for a given finite input, one output; this paradigm does not do justice to the ongoing interactions between a reactive system and its environment. To add ongoing interactivity to Turing machines, interaction machines are proposed in [62], and formalised in [31] as persistent Turing machines. Likewise, [5] proposes reactive Turing machines. Since standard process algebras like CCS are already equipped with interaction primitives, they can surely also model computations on persistent or reactive Turing machines. All this strengthens the feeling that standard process algebras, such as CCS, are powerful enough to specify any distributed system.
2 Fairness Assumptions

Before presenting evidence that CCS and related formalisms cannot correctly specify every distributed system, some explanation is in order on our understanding of ‘correctly’. This is best illustrated by an example.

Consider the CCS agent identifier \( E \) with defining equation \( E \overset{\text{def}}{=} a.E + b.0 \). The question is whether this is a good rendering of a process that is guaranteed to eventually perform the action \( b \). The answer depends on whether we incorporate a fairness assumption in the semantics of CCS. A strong fairness assumption requires that if a task (here \( b \)) is enabled infinitely often, but allowing interruptions during which it is not enabled, it will eventually be scheduled \([23, 42]\). Making such an assumption allows us to infer that indeed the process \( E \) will eventually do a \( b \).

It depends on the context of the application of CCS whether it is appropriate to make such fairness assumptions. For the verification of the alternating bit protocol, for instance, fairness assumptions are indispensable \([3]\). But in some situations they allow us to reach conclusions that are patently false. In the example above for instance, let \( a \) be an unsuccessful attempt to dial a number or an unreliable mobile phone, and \( b \) a successful one. The system \( E \) simply retries after each unsuccessful attempt. Whether it ever succeeds in performing \( b \) depends very much on how unreliable the phone is. If there is a fixed positive probability on success, the strong fairness assumption appears warranted. Yet, if the phone is completely dead, it is not, and the conclusion that we eventually succeed in dialling is false. In fact, when assuming strong fairness we loose the expressiveness to describe by a finite recursive specification like \( E \) a system such as the above interaction with the unreliable telephone that \textit{does} allow an infinite run with only \( a \).

As evidence that not every distributed system can be rendered correctly in CCS, we will describe a \textit{fair scheduler} as a counterexample. Yet, our fair scheduler can be rendered in CCS very easily, if only we are willing to postulate a fairness property to support its correctness. However, considering the above example and the fact that we may reach wrong conclusions, this is a price we are not willing to pay.

Our fair scheduler is not merely an ‘artificial’ CCS specification; it is implemented in many working distributed systems, and (unlike the alternating bit protocol) its correctness should not be contingent on any fairness assumption whatsoever. This is another reason why we do not want to invoke fairness to achieve a correct rendering in CCS.

Yet, we do find it reasonable to equip CCS with two assumptions that are weaker than strong fairness, namely progress and justness. A progress assumption is what is needed to infer that the process \( b.0 \) eventually do a \( b \), and a justness assumption allows us to infer that the parallel composition \( A|b.0 \) with \( A \overset{\text{def}}{=} a.A \) will eventually do a \( b \). If our task is to specify in CCS a process \( B \) that will eventually do a \( b \), then, when assuming strong fairness, the processes \( E, A|b.0 \) and \( b.0 \) are acceptable implementations of \( B \). If we assume justness, but not fairness, this selection shrinks to \( A|b.0 \) and \( b.0 \), and if we only assume progress, we have to give up on \( A|b.0 \) as well. When not even assuming progress, \( B \) cannot be rendered in CCS at all. Assuming progress and justness only, \( A|b.0 \) models a process that

---

1 In \([1]\) a form of reasoning using a particularly strong global fairness assumption was integrated in the axiomatic framework of ACP, and shown to be compatible with the notion of weak bisimulation commonly taken as the semantic basis for CCS.
will eventually do a b, whereas $E$ can be used to characterise the above mentioned interaction with the unreliable telephone, which allows an infinite sequence of as only.

So, when we claim that a fair scheduler cannot be implemented in CCS, we mean that it cannot be implemented in CCS+justness, CCS+progress or CCS without any progress assumption. It can be implemented in CCS+strong fairness, however.

To pinpoint the borderline, consider a weak fairness or justice assumption \[25, 42\]. This assumption requires that if a task, from some point onwards, is perpetually enabled, it will eventually be scheduled. What this means depends on our interpretation of ‘perpetual’. If ‘perpetual’ simply means ‘in each state’, then a weak fairness assumption is all that is needed to assure that the process $E$ will eventually do a b\footnote{The process $E'$ with $E' \equiv a.a.E' + b.0$ on the other hand really needs strong fairness.}. A weak fairness assumption in this sense is enough to correctly render a fair scheduler in CCS. If, on the other hand, the execution of the $a$-transition of $E$ counts as a (short) interruption of the enabledness of $b$, then justice can be shown to coincide with justness \[29\]; as we will show, this is not enough to render a fair scheduler in CCS.

3 Specifications versus Actual Processes

Consider the system specification $G$ expressed in CCS as $A|B$ with $A \equiv a.A$ and $B \equiv b.B$, but with the added requirement that all infinite executions should have infinitely many occurrences of $a$ as well as $b$. Here $a$ and $b$ could be seen as two tasks that need to be scheduled again and again. The left-hand component $A$ of the parallel composition tries to perform task $a$ infinitely often, and the right-hand component tries to perform task $b$ infinitely often. The process $A|B$ by itself, as specified in CCS, is normally deemed equivalent to the process $C$, defined by $C \equiv a.C + b.C$, and—in the absence of a justness or fairness assumption—offers no guarantee that a single $b$ will ever happen. It could be that, due to unfortunate scheduling, at each time a choice is made, task $a$ is chosen. The challenge in specifying the fair version $G$ of this process in CCS is how to ensure that sooner or later a $b$ will happen, without simply invoking a fairness or justness assumption, and without setting any fixed limit on the number of $a$s that can happen beforehand.

Accordingly, solutions have appeared in the literature that change the operational semantics of CCS in such a way that $A|B$ will surely do a $b$ eventually. In \[52\] for instance, parallel operators $∥_m$ are used that, each time a $b$ occurs, nondeterministically select a number $m$ and guarantee that from that point onwards at most $m$ occurrences of $a$ happen before the next $b$. Another solution along these lines is proposed in \[17\], whereas \[13\] solves the problem by harvesting the power added by the treatment of time in the timed process algebra PAFAS \[16\].

In relation to the above challenge it would be trivial to specify some process that makes sure that tasks $a$ and $b$ are each scheduled infinitely often; a particularly simple way to achieve this is through the CCS specification $D$, given by $D \equiv a.b.D$; that is, to alternate each of the two tasks. This is a round-robin solution. It could be seen as a particular implementation of $G$. The reason that such a
solution is not chosen is that it fails to capture the full generality of the original specification, in which arbitrary many \(a\)s may come between any two \(b\)s.

Any real-life implementation of \(G\) on a physical computer is unlikely to capture the full generality of its specification, but rather goes a few steps towards the round-robin solution. For this reason, one could argue that \(G\) does not constitute an example of a distributed system that cannot be rendered in CCS, but rather one of a specification that cannot be rendered in CCS. As such, it falls out of the scope of this paper.

4 Our Contributions

This paper counters the sentiment that CCS-like process algebras are powerful enough to represent arbitrary distributed systems by presenting a particularly simple system of which no implementation can be expressed in CCS. The reason we use CCS is that it is among the most well-known standard process algebras, while having a fairly easy to explain syntax and semantics. However, we believe the same result, with essentially the same proof, could be obtained for COSY [40], CSP [11,36], ACP [7], LOTOS [9], \(\mu\)CRL [33], the \(\pi\)-calculus [46], etc.

Our system is a fair scheduler. It can receive two kinds of requests \(r_1\) and \(r_2\) from its environment on separate channels, and is guaranteed to perform a task—granting the request—in response. Our fairness requirement rules out a scheduler that may fail to ever grant a request of type \(r_1\) because it is consistently busy granting requests \(r_2\).

Such schedulers occur (in suitable variations) in many distributed systems. Examples are First in First out [3], Round Robin, and Fair Queueing scheduling algorithms as used in network routers [17,18] and operating systems [38], or the Completely Fair Scheduler [5] which is the default scheduler of the Linux kernel since version 2.6.23.

If \(F\) stands for the most general specification of our scheduler, our claim entails that \(F\) cannot be rendered in CCS. However, accurately expressing \(F\) in CCS can be seen as a luxury problem. Here we would accept any implementation of \(F\), under the broadest definition of implementation that makes sense for this problem—a round-robin solution for instance would be totally acceptable—and what we show is that even that is impossible.

As is common, we employ a version of CCS that allows the use of arbitrary sets of recursive equations to define processes. As is trivial to show, any labelled transition system, computable or not, can be modelled up to strong bisimulation equivalence as an expression in this language. Hence, our result implies that no implementation of the fair scheduler \(F\) can be modelled as a labelled transition system modulo strong bisimulation equivalence.

In this paper we will use a semantics of CCS incorporating a justness assumption. It distinguishes the strongly bisimilar systems \(A|B\) and \(C\) mentioned above, on grounds that \(A|B\) can be understood to always perform infinitely many \(a\)s as well as \(b\)s, whereas \(C\) might perform an infinite sequence of \(b\)s while discarding the

\(^3\) Also known as First Come First Served (FCFS)

\(^4\) [http://en.wikipedia.org/wiki/Scheduling_(computing)]

\(^5\) [http://en.wikipedia.org/wiki/Completely_Fair_Scheduler]
a-option all the time. This semantics increases the power of CCS in specifying fair schedulers, and thereby strengthens our result that no implementation of the fair scheduler $F$ can be expressed. It thereby becomes stronger than the result that no implementation of $F$ can be rendered as a labelled transition system modulo strong bisimulation equivalence.

To prove our result, we show that our fair scheduler cannot be expressed in terms of safe Petri nets. The result for CCS then follows by reduction: an adequate Petri net semantics of CCS shows that if the scheduler could be expressed in CCS, it could also be expressed as a Petri net.

The reason we resort to Petri nets to prove our main theorem is that Petri nets offer a structural characterisation of what it means for a transition to be continuously enabled in a run of the represented system from some state onwards. This is exploited in the proofs of Lemmas 1–4. It would be much harder to prove their counterparts directly in terms of the labelled transition system of CCS.

In different formulations, our impossibility result for Petri nets was established earlier by Vogler in [60] and by Kindler & Walter in [37], but in both cases side conditions were imposed that inhibit lifting these results to CCS. The proof of [60, Lemma 6.1] considers only finite Petri nets. The argument would extend to finitely branching nets, but not to all Petri nets that arise as the semantics of CCS expressions. The proof of [37] is restricted to Petri nets that interact with their environment through an interface of a particular shape, and it is not a priori clear that this does not cause a loss of generality. However, in Section 13 we study a similar interface in the context of CCS and show that it does not limit generality.

Although our fair scheduler cannot be expressed in a standard process algebra like CCS, we believe there are many extensions in the literature in which this can be done quite easily. In Section 11 for instance, we specify it in a formalism that could be called CCS+LTL. The use of a priority operator appears to be sufficient as well.

### 5 Peterson’s and Dekker’s Mutual Exclusion Protocols

Since Peterson’s and Dekker’s mutual exclusion protocols yield instances of our fair scheduler, it follows that these protocol cannot be rendered correctly in CCS without imposing a fairness assumption. Nevertheless, implementations of these algorithms in CCS or similar formalisms occur frequently in the literature, and almost never a fairness assumption is invoked. Moreover, for each of these two protocols, its various renderings differ only in insignificant details. Our result implies that these common renderings cannot be correct. Usually, only safety properties of these protocols are shown: never are two processes simultaneously in the critical section. The problem is with the liveness property: any process that is ready to enter the critical section will eventually do so. We found four papers that claim to establish essentially this property, of which only one invokes a fairness assumption. We will indicate in which way the other three do not establish the right liveness property.

Peterson expressed his protocol correctly in pseudocode without resorting to a fairness assumption, although progress and justness are assumed implicitly. It follows that Peterson’s pseudocode does not admit an accurate translation into CCS. We pinpoint the problem in this paper.
6 Overview

In Part I we discussed (informally) the goal we set out to achieve, and why we believe it is important and surprising at the same time.

Part II formalises our results, while providing explanations of the choices made in this formalisation. In particular, Section 7 presents an informal description of our fair scheduler $F$. Section 8 presents CCS. Section 9 makes a progress assumption on the semantics of CCS and argues that it is useful to set apart a set of non-blocking actions. Section 10 formalises the justness assumption discussed above and presents a semantics of CCS in which a process $P$ is modelled as a state in a labelled transition system together with a set of (possibly infinite) paths in that transition system starting from $P$ that model its valid runs. Section 11 gives a formal specification of $F$. Since we aim to show that no implementation of $F$ can be specified in CCS, the specification of $F$ cannot be given in CCS either. Instead we specify $F$ as a CCS expression augmented with a fairness specification. This follows the traditional approach of TLA [39] and other formalisms [24], “in which first the legal computations are specified, and then a fairness notion is used to exclude some computations which otherwise would be legal” [2]. In Section 12 we state our main result, saying that no fair scheduler—that is: no implementation of $F$—can be expressed in CCS. Section 13 reformulates this result, so that it is independent of the concept of an action being perpetually enabled in a run of the represented system. In Section 14 we conclude that mutual exclusion protocols, like the algorithms from Dekker or Peterson, cannot be rendered correctly in CCS without imposing a fairness condition. We also investigate the apparent contradiction with the fact that several research papers claim to achieve exactly this. We end this section with a result by Corradini, Di Berardini & Vogler, showing where a fairness assumption is needed for a rendering of Dekker’s protocol in a process algebra to be correct.

Part III deals with proving our main result. In Section 15 we formulate our claim that no fair scheduler can be modelled as a safe Petri net. This claim is proven in Section 16. In Section 17 an operational Petri net semantics of CCS is presented, following the work of Degano, De Nicola & Montanari. From this, the proof of our main result is obtained in Section 18. A few concluding remarks are made in Section 19.

Part II Formalisation

7 A Fair Scheduler

Our fair scheduler is a reactive system with two input channels: one on which it can receive requests $r_1$ from its environment and one on which it can receive requests $r_2$. We allow the scheduler to be too busy shortly after receiving a request $r_1$ to accept another request $r_1$ on the same channel. However, the system will always return to a state where it remains ready to accept the next request $r_1$ until $r_1$ arrives. In case no request arrives it remains ready forever. The environment is under no obligation to issue requests, or to ever stop issuing requests. Hence for any numbers $n_1$ and $n_2 \in \mathbb{N} \cup \{\infty\}$ there is at least one run of the system in which exactly that many requests of type $r_1$ and $r_2$ are received.
Every request \( r_i \) asks for a task \( t_i \) to be executed. The crucial property of the fair scheduler is that it will eventually grant any such request. Thus, we require that in any run of the system each occurrence of \( r_i \) will be followed by an occurrence of \( t_i \). In Linear-time Temporal Logic (LTL) [53] this can be stated as

\[
G(r_i \Rightarrow F(t_i)), \quad i \in \{1, 2\}.
\]

Note that it may happen that the environment issues request \( r_1 \) three times in a row before the scheduler got a chance to schedule task \( t_1 \). In that case the scheduler may fulfil its obligation by scheduling task \( t_1 \) just once. Hence it need not keep a counter of outstanding requests.\(^6\)

We are not interested in implementations of the scheduler that just schedule both tasks infinitely often without even listening to the requests. Hence we require that in any partial run of the scheduler there may not be more occurrences of \( t_i \) than of \( r_i \), for \( i = 1, 2 \).

The last requirement is that between each two occurrences of \( t_i \) and \( t_j \) for \( i, j \in \{1, 2\} \) an intermittent activity \( e \) is scheduled.\(^7\) This requirement will rule out fully concurrent implementations, in which there are parallel components for task \( t_1 \) and task \( t_2 \) that do not interact in any way.

### 8 The Calculus of Communicating Systems

The Calculus of communicating systems (CCS) [45] is parametrised with sets \( A \) of names and \( \mathcal{K} \) of agent identifiers; each \( A \in \mathcal{K} \) comes with a defining equation \( A \overset{\text{def}}{=} \alpha.P \) with \( P \) being a CCS expression as defined below. The set \( \mathcal{A} \) of co-names is \( \mathcal{A} := \{\bar{a} | a \in A\} \), and the set \( \mathcal{H} \) of handshake actions is \( \mathcal{H} := \mathcal{A} \cup \mathcal{A} \), the disjoint union of the names and co-names. The function \( \cdot \) is extended to \( \mathcal{H} \) by declaring \( \bar{a} := a \). Finally, \( \mathcal{A} : = \mathcal{H} \cup \{\tau\} \) is the set of actions. Below, \( a, b, c \) range over \( \mathcal{H} \), \( \alpha, \beta \) over \( \mathcal{A} \) and \( A \) over \( \mathcal{K} \). A relabelling is a function \( f : \mathcal{H} \rightarrow \mathcal{H} \) satisfying \( f(\bar{a}) = f(a) \); it extends to \( \mathcal{A} \) by \( f(\alpha) := \tau \). The set \( \text{TCCS} \) of CCS expressions is the smallest set including:

\[\alpha.P \overset{\alpha}{\rightarrow} P, \sum_{i \in I} P_i \overset{\alpha}{\rightarrow} P' \]

\[P \overset{\alpha}{\rightarrow} P', P(Q \overset{\alpha}{\rightarrow} P'|Q) \]

\[P(Q \overset{\alpha}{\rightarrow} P'|Q') \]

\[Q \overset{\alpha}{\rightarrow} Q' \]

\[P \overset{\alpha}{\rightarrow} P' \quad (a \neq \alpha \neq \bar{a}) \]

\[P[\alpha] \overset{\alpha}{\rightarrow} P'[\alpha] \]

\[P[\tau \overset{\alpha}{\rightarrow} P'[\tau] \]

\[A \overset{\alpha}{\rightarrow} P', A \overset{\alpha}{\rightarrow} P' \]

\[P \overset{\alpha}{\rightarrow} P' \quad (A \overset{\text{def}}{=} \alpha.P) \]

Table 1 Structural operational semantics of CCS

---

\(^6\) This relaxed requirement only serves to increase the range of acceptable schedulers, thereby strengthening our impossibility result. It by no means rules out a scheduler that schedules task \( t_1 \) exactly once for each request \( r_1 \) received.

\(^7\) Our specification places no restrictions on the presence or absence of any additional occurrences of \( e \). This again increases the range of acceptable implementations.
for $P, P_i, Q \in T_{\text{CCS}}$, index sets $I$, and relabellings $f$. We write $P_1 + P_2$ for $\sum_{i \in I} P_i$ if $I = \{1, 2\}$, and 0 if $I = \emptyset$. The semantics of CCS is given by the labelled transition relation $\to \subseteq T_{\text{CCS}} \times \text{Act} \times T_{\text{CCS}}$, where the transitions $P \xrightarrow{\alpha} Q$ are derived from the rules of Table 1. The pair $(T_{\text{CCS}}, \to)$ is called the labelled transition system (LTS) of CCS.

9 The Necessity of Output Actions

Before attempting to specify our scheduler in CCS, let us have a look at a simpler problem: the same scheduler, but with only one type of request $r$, and one type of task $t$ to be scheduled. A candidate CCS specification of such a scheduler is the process $F_0$, defined by

$$F_0 \overset{\text{def}}{=} r.e.t.F_0.$$

As stated in Section 7, the scheduler is called fair if every request $r$ is eventually followed by the requested task $t$; so we want to ensure the property $G(r \Rightarrow F(t))$.

However, we cannot guarantee that this property actually holds for process $F_0$. The reason is that the process might remain in the state $s$ reached by taking transition $e$ without ever performing the action $t$. In any formalism that allows to remain in a state even when there are enabled actions, no useful liveness property about processes can ever be guaranteed. One therefore often makes a progress assumption, saying that the system will not idle as long as it can make progress. Armed with this assumption, it appears fair to say that the process $F_0$ satisfies the required property $G(r \Rightarrow F(t))$.

However, by symmetry, the same line of reasoning would allow us to derive that $F_0$ satisfies $G(t \Rightarrow F(r))$, i.e. each execution of $t$ will be followed by a new request. Yet, this is something we specifically do not want to assume: the action $r$ is meant to be fully under the control of the environment, and it may very well happen that at some point the environment stops making further requests. A particular instance of this is when the environment is modelled by a CCS context such as $(\bar{r}0\bar{t}\bar{r}0)\bar{t}$; in this context the process $F_0$ will receive the request $r$ only once.

Hence, we reject the validity of $G(t \Rightarrow F(r))$ based on environments such as $(\bar{r}0\bar{t}\bar{r}0)\bar{t}$; however, the same reasoning allows environments such as $(\bar{r}0\bar{t}\bar{r}0)\bar{r}\bar{t}$ that do not allow the task $t$ to be executed more than once. The existence of such environments totally defeats our scheduler, or any other one.

Thus, for a fair scheduler to make sense, we need to consider environments that have full control over the action $r$ but cannot sabotage the mission of our scheduler by disabling tasks $t$ and $e$. We formalise this by calling $t$ and $e$ output actions. An output action [23, Section 9.1] is an activity of our system that cannot be stopped by its environment; or, equivalently, considering an action $t$ to be output means that we choose not to consider environments that can block $t$. In our schedulers, $t$ and $e$ are output actions, whereas $r$ is not.

8 When assuming that this formula holds, $F_0$ trivially satisfies the other properties required in Section 7 the system will always return to a state where it remains ready to accept the next request $r$ until it arrives; in any partial run there are no more occurrences of $t$ than of $r$, and between each two occurrences of $t$ the action $e$ is scheduled.
Let CCS$^1$ be the variant of CCS that is parametrised not only by sets $\mathcal{A}$ of names and $\mathcal{K}$ of agent identifiers, but also by a set $\mathcal{O}$ of output actions. The only further difference with CCS of Section 8 is that $\text{Act} := \mathcal{K} \cup \mathcal{O} \cup \{\tau\}$, and a relabelling $f$ extends to $\text{Act}$ by $f(\alpha) := \alpha$ for all $\alpha \in \mathcal{K} \cup \{\tau\}$. CCS$^1$ can be seen as an extension of CCS with output actions, but it can just as well be seen as a restriction of CCS in which for some of the names there are no co-names and no restriction operators.

It should be noted that CCS already features the concept of an internal action, namely $\tau$, of which it is normally assumed that it cannot be blocked by the environment. Yet, for the purpose of specifying our scheduler, the rôle of the output action $t$ cannot be played by $\tau$, for the internal action is supposed to be unobservable and is easily abstracted away. Output actions share the feature of internal actions that whether they occur or not is determined by the internal work of the specified system only; yet at the same time they are observable by the environment in which the system is running. An action in $\mathcal{K} \cup \{\tau\}$—so an output or internal action—is also called a non-blocking action.

Now we formulate our progress assumption:\[23,29]:

Any process in a state that admits a non-blocking action will eventually perform an action.

LTL formulas are deemed to hold for a process $P$ if they hold for all complete paths of $P$ in the labelled transition system of CCS$^1$. Here a path of $P$ is an alternating sequence of states and transitions, starting from the state $P$ and either being infinite or ending in a state, such that each transition in the sequence goes from the state before to the state after it, and a finite path is complete iff it does not end in a state that enables a non-blocking action; completeness of infinite paths is discussed in the next sections. For further details, see [23, Section 9.1] or [29].

Assuming progress, the scheduler $F_0 \overset{\text{def}}{=} \text{r.e.t.} F_0$ satisfies $G(\tau \Rightarrow F(t))$ because on each complete path every $\tau$ is followed by a $t$. Hence $F_0$ is fair w.r.t. the simpler problem.

10 A Just Semantics of Parallelism

In the previous section we considered a scheduler that was significantly simpler than the one of Section 7 and were able to specify it in CCS$^1$ by $F_0 \overset{\text{def}}{=} \text{r.e.t.} F_0$, with output actions $t$ and $e$. However, in order to ensure that our specification was formally correct, we needed to introduce the concept of an output action, and made a progress assumption on the semantics of the language.

In this section, we consider again a simplification of the scheduler of Section 7 and once more succeed in specifying it in CCS$^1$. Again we need to make an assumption about the semantics of CCS$^1$ in order to ensure that our specification is formally correct.

\[9\] The output and internal actions of CCS$^1$ are similar to the output and internal action of I/O automata [44]. However, the remaining actions of I/O automata are input actions that are totally under the control of the environment of the modelled system. In CCS, on the other hand, the default type of action is a synchronisation that can happen only in cooperation between a system and its environment.
Both assumptions increase the range of correct CCS specifications and thereby make the promised result on the absence of any CCS specification of a scheduler as described in Section 7 more challenging.

Consider a scheduler as described in Section 7 but without the last requirement about the intermittent activity $e$. A candidate CCS specification is the process $F_1 | F_2$, defined by

$$F_i = r_i . t_i . F_i , \quad i \in \{1, 2\}.$$  

Here, and throughout this paper, $t_i$ (like $e$) is an output action and $r_i$ is not. For this scheduler to be fair, it has to satisfy $G(r_i \Rightarrow F(t_i))$ for $i = 1, 2$  

By the reasoning of the previous section the process $F_i$ satisfies the temporal formula $G(r_i \Rightarrow F(t_i))$ for $i = 1, 2$. It is tempting to conclude that obviously their parallel composition $F_1 | F_2$ satisfies both of these requirements. Yet, the system run $r_1 (r_2 t_2)^\omega$—that after performing one action from $F_1$ performs infinitely many actions from $F_2$ without interleaving any further actions from $F_1$—could be considered a counterexample.

Here we take the point of view that no amount of activity of $F_2$ can prevent $F_1$ from making progress. The system $F_1 | F_2$ simply does not have a run $r_1 (r_2 t_2)^\omega$. The corresponding path from the state $F_1 | F_2$ in the LTS of CCS is no more than an artifact of the use of interleaving semantics. In general, we make the following justness assumption \[20\]:

If a combination of components in a parallel composition is in a state that admits a non-blocking action, then one (or more) of them will eventually partake in an action.

Thus justness guarantees progress of all components in a parallel composition, and of all combinations of such components. In the CCS expression $( (P|Q ) \backslash a ) | R$ with $P \overset{\text{def}}{= } a . P + c . P$, $Q \overset{\text{def}}{= } a . Q$ and $R \overset{\text{def}}{= } b . R + \bar{c} . R$ for instance there is a state where $P$ admits an action $c \in \mathcal{X}$ with $c \neq a$ and $R$ admits an action $\bar{c}$. Thereby, the combination of these components admits an action $\tau$. Our justness assumption now requires that either $P$ or $R$ will eventually partake in an action. This could be the $\tau$-action obtained from synchronising $c$ and $\bar{c}$, but it also could be any other action involving $P$ or $R$. In each case the system will (at least for an instant) cease to be in a state where that synchronisation between $P$ and $R$ is enabled. Note that progress is a special case of justness, obtained by considering any process as the combination of all its parallel components.

In \[20\] we formalised the justness assumption as follows. Any transition $P|Q \xrightarrow{\alpha} R$ derives, through the rules of Table 1, from

- a transition $P \xrightarrow{\alpha} P'$ and a state $Q$, where $R = P'|Q$,
- two transitions $P \xrightarrow{\alpha} P'$ and $Q \xrightarrow{\beta} Q'$, where $R = P'|Q'$,
- or from a state $P$ and a transition $Q \xrightarrow{\alpha} Q'$, where $R = P|Q'$.

This transition/state, transition/transition or state/transition pair is called a decomposition of $P|Q \xrightarrow{\alpha} R$; it need not be unique, as we will show in Example 1 below. Now a decomposition of a path $\eta$ of $P|Q$ into paths $\eta_1$ and $\eta_2$ of $P$ and $Q$, \[10\] when assuming that these formulas hold, $F_1 | F_2$ trivially satisfies the other properties required of it: the system will always return to a state where it remains ready to accept the next request $r_i$ until it arrives—hence for any numbers $n_1$ and $n_2 \in \mathbb{N} \cup \{\infty\}$ there is at least one run of the system in which exactly that many requests of type $r_1$ and $r_2$ are received—and in any partial run there are no more occurrences of $t_i$ than of $r_i$. 

---

10 When assuming that these formulas hold, $F_1 | F_2$ trivially satisfies the other properties required of it: the system will always return to a state where it remains ready to accept the next request $r_i$ until it arrives—hence for any numbers $n_1$ and $n_2 \in \mathbb{N} \cup \{\infty\}$ there is at least one run of the system in which exactly that many requests of type $r_1$ and $r_2$ are received—and in any partial run there are no more occurrences of $t_i$ than of $r_i$. 

---

CCS: It’s not Fair!
respectively, is obtained by decomposing each transition in the path, and concatenating all left-projections into a path of P and all right-projections into a path of Q. Here it could be that η is infinite, yet either η₁ or η₂ (but not both) are finite. Again, decomposition of paths need not be unique.

Similarly, any transition $P[f] \xrightarrow{\sigma} R$ stems from a transition $P \xrightarrow{\sigma} P'$, where $R = P'[f]$ and $\alpha = f(\beta)$. This transition is called a decomposition of $P[f] \xrightarrow{\sigma} R$. A decomposition of a path η of $P[f]$ is obtained by decomposing each transition in the path, and concatenating all decompositions so obtained into a path of P. A decomposition of a path of $P \setminus c$ is defined likewise.

**Definition 1** Y-justness, for $Y \subseteq \mathcal{H}$ is the largest family of predicates on the paths in the LTS of CCS\(^1\) such that

- a finite Y-just path ends in a state that admits actions from Y only;
- a Y-just path of a process $P|Q$ can be decomposed into an X-just path of P and a Z-just path of Q such that $Y \supseteq X \cup Z$ and $X \cap Z = \emptyset$—here $Z := \{ \bar{c} \mid c \in Z \}$;
- a Y-just path of $P \setminus c$ can be decomposed into a $Y \cup \{ \bar{c} \}$-just path of P;
- a Y-just path of $P[f]$ can be decomposed into an $f^{-1}(Y)$-just path of P;
- and each suffix of a Y-just path is Y-just.

A path η is just if it is Y-just for some $Y \subseteq \mathcal{H}$. It is a-enabled for an action $a \in \mathcal{H}$ if $a \in Y$ for all Y such that η is Y-just.

Intuitively, a Y-just path models a run in which Y is an upper bound of the set of labels of abstract transitions\(^2\) that from some point onwards are continuously enabled but never taken. Here an abstract transition with a label from $\mathcal{H}$ is deemed to be continuously enabled but never taken iff it is enabled in a parallel component that performs no further actions. Such a run can actually occur if the environment from some point onwards blocks the actions in Y.

The last clause in the second requirement prevents an X-just path of P and a Z-just path of Q to compose into an X∪Z-just path of $P|Q$ when X contains an action $a$ and Z the complementary action $\bar{a}$. The reason is that no environment can block both actions for their respective components, as nothing can prevent them from synchronising with each other. The fifth requirement helps characterising processes of the form $b.0 + (A|b.0)$ and $a.(A|b.0)$, with $A \equiv a.A$. Here, the first transition ‘gets rid of’ the choice and of the leading action $a$, respectively, and reduces the justness of paths of such processes to their suffixes.

A complication in understanding Definition 1 is that a single path could be seen as modelling different system runs of which one could be considered just, respectively a-enabled, and the other not.

**Example 1** Consider the process $B|B$ defined by $B \overset{a}{\rightarrow} b.B$. The only transition of this process is $B|B \xrightarrow{b} B|B$, so $B|B$ has exactly one infinite path η, obtained by repeating this transition infinitely often. Assuming that $b$ is an output action, one may wonder if η should count as being just. In case all transitions in η originate from the left component, the $b$-transition of the right component is continuously enabled but never taken. This does not correspond to a (just) run of the represented

\(^1\) By definition Y does not contain non-blocking action.

\(^2\) The CCS process $a.0|b.0$ has two transitions labelled $a$, namely $a.0|b.0 \xrightarrow{a} 0|b.0$ and $a.0|0 \xrightarrow{a} 0|0$. The only difference between these two transitions is that one occurs before the action $b$ is performed by the parallel component and the other afterwards. In \([29]\) we formalise a notion of an abstract transition that identifies these two concrete transitions.
system. However, in case $\eta$ alternates transitions from each component, it does model a (just) run. The mere fact that a $b$-transition is enabled on every state of $\eta$ has no bearing on the matter. Now Definition 11 considers $\eta$ just, on grounds of the fact that it models some (just) run.

If in this example $b$ is a handshake action, the path $\eta$ models a (just) run in which a $b$-labelled abstract transition is continuously enabled but never taken; but it also models a (just) run in which no transition is continuously enabled but never taken. According to Definition 11, $\eta$ counts as $\emptyset$-just, and thus is not deemed $b$-enabled. Intuitively, a path is $b$-enabled iff on all runs modelled by that path a transition labelled $b$ is continuously enabled but never taken.

Now a just path, as defined above, is our default definition of a complete path, as contemplated at the end of Section 9. Indeed, a finite path is just iff it does not end in a state from which a non-blocking action is possible [29]. Thus, our semantics of a CCS! process $P$ consists of the state $P$ in the LTS of CCS! together with the set of complete paths in that LTS starting from $P$ [23, 29]. LTL formulas hold for $P$ iff they are valid on all complete paths of $P$.

Here we employ a just semantics of CCS! by taking the just paths to be the complete ones. This way $F_0$ is a correct specification of the scheduler required in Section 9 and $F_1 | F_2$ is a correct specification of the scheduler required above.

11 Formal Specification of the Fair Scheduler

We now provide a formal specification of the scheduler described in Section 7. Since the aim of this paper is to show that this cannot be done in CCS! (and thus certainly not in CCS) we need a different formalism for this task. Here we follow the traditional approach of TLA [39] and several other frameworks [24], “in which first the legal computations are specified, and then a fairness notion is used to exclude some computations which otherwise would be legal” [2]. Following [29], we use CCS! for the first step and LTL for the second.

Thus, in this section we specify a process as a pair of a CCS! expression $P$ and a set $\mathcal{F}$ of LTL formulas, called a fairness specification. The semantics of $P$ consists of the state $P$ in the LTS of CCS! together with the set of just paths in that LTS starting from $P$. The formulas of $\mathcal{F}$ are evaluated on the paths of $P$ and any path that satisfies all formulas of $\mathcal{F}$ is called fair. Now the semantics of the entire specification $(P, \mathcal{F})$ is the state $P$ in the LTS of CCS! together with the set of complete paths of $P$, defined as those paths that are both just and fair. In [23, 29] a consistency requirement is formulated that should hold between $P$ and $\mathcal{F}$.

Now a fair scheduler as described in Section 7 can be specified by the CCS! process $(I_1 | G \mid I_2) \backslash c_1 \backslash c_2$, where

$I_i \stackrel{\text{def}}{=} r_i, c_i, I_i$ \quad (i \in \{1, 2\}) \quad \text{and} \quad G \stackrel{\text{def}}{=} c_1, t_1, e, G + c_2, t_2, e, G,$

augmented with the fairness specification $\bigwedge_{i=1,2} G (r_i \Rightarrow F(t_i)).$

Here the first requirement of Section 7 is satisfied by locating the two channels receiving the requests $r_1$ and $r_2$ on different parallel components $I_1$ and $I_2$. This way, after performing, say, $r_1$, the system—component $I_1$ to be precise—will always return to a state where it remains ready to accept the next request $r_1$ until it arrives, independent of occurrences of $r_2$. The (non-output) actions $c_i$ are used to
communicate the request from the processes $I_i$ to a central component $G$, which then performs the requested action $t_i$.

The second requirement of Section 7 is enforced by the fairness specification, and the last two requirements of Section 7 are met by construction: in any partial run there are no more occurrences of $t_i$ than of $r_i$, and between each two occurrences of $t_i$ and $t_j$ for $i, j \in \{1, 2\}$ the intermittent action $e$ is scheduled.

12 Fair Schedulers Cannot be Rendered in CCS$^f$—Formalisation

In this section we formulate the main result of the paper, namely that no scheduler as described in Sections 7 and 11 can be specified in CCS$^f$. Since we already showed that it can be specified in CCS$^f$ augmented with a fairness specification, here, and in the rest of the paper, we confine ourselves to CCS$^f$ without fairness specifications. Thus, our notion of a complete path is (re)set to that of a just path, as specified in Definition 1.

Theorem 1 There does not exist a CCS$^f$ expression $F$ such that:

1. any complete path of $F$ that has finitely many occurrences of $r_i$ is $r_i$-enabled;  
2. on each complete (i.e., just) path of $F$, each $r_i$ is followed by a $t_i$;  
3. on each finite path of $F$ there are no more occurrences of $t_i$ than of $r_i$; and  
4. between each two occurrences of $t_i$ and $t_j$ ($i, j \in \{1, 2\}$) an action $e$ occurs.

Requirements 1–4 exactly formalise the four requirements described in Section 7. We proceed to show that none of them can be skipped.

The CCS$^f$ process $F_1|F_2$ of Section 10 satisfies Requirements 1, 2 and 3. It does not satisfy Requirement 4, due to the partial run $r_1t_2t_1t_2$.

The CCS$^f$ process $E_1|G_1|E_2$ with $E_i \overset{df}{=} r_i.E_i$ for $i = 1, 2$ and $G_1 \overset{df}{=} t_1.e.t_2.e.G_1$ satisfies Requirements 1, 2 and 4. It does not satisfy Requirement 3 due to the partial run consisting of the single action $t_1$.

The CCS$^f$ process $E_1.E_2$ satisfies Requirements 1, 3 and 4, but not 2.

Finally, the process $G_2$ given by $G_2 \overset{df}{=} r_1.t_1.e.G_2 + r_2.t_2.e.G_2$ satisfies Requirements 2, 3 and 4. However, it does not satisfy Requirement 1 because it allows the $\emptyset$-just path $(r_2t_2e)\omega$ with no occurrences of $r_1$. This path models a run in which the system never reaches a state where it remains ready to accept the next request $r_1$.

The proof of Theorem 1 will be the subject of Part III.

13 A Characterisation of Fair Schedulers without $a$-enabling

Below we will show that without loss of generality we may assume any fair scheduler to have a specific form. If it has that form, Requirement 1 is redundant. Hence Requirement 1 can be replaced by requiring that the scheduler is of that form. This variant of our result appeared as a conjecture in [29].

For any CCS$^f$ expression $F$, let $\bar{F} := (I_1 \mid F[f] \mid I_2)\backslash c_1 \backslash c_2$ with $I_i \overset{df}{=} r_i.e_i.I_i$ for $i \in \{1, 2\}$, where $f$ is an injective relabelling with $f(r_i) = c_i$ for $i = 1, 2$, and $r_1, r_2, r_1, r_2 \notin f(X)$. By the definition of relabelling (cf. Section 9), $f(t_i) = t_i$ and $f(e) = e$. 
Theorem 2 A process \( F \) meets Requirements [1][4] of Theorem [1] iff \( \tilde{F} \) meets these requirements, which is the case iff \( \tilde{F} \) meets Requirements [2][4].

Proof Suppose \( F \) satisfies Requirements [1][4]

1. To show that \( \tilde{F} \) satisfies Requirement [1] (with \( i := 1 \); the other case follows by symmetry), it suffices to show that each occurrence of \( r_1 \) in a just path of \( \tilde{F} \), which corresponds to an occurrence of \( r_1 \) in the subprocess \( I_1 \), is followed by an occurrence of \( c_1 \) in \( I_1 \).

   So assume, towards a contradiction, that on a just path \( \eta \) of \( \tilde{F} \) an occurrence of \( r_1 \) is not followed by an occurrence of \( c_1 \) in the subprocess \( I_1 \). By Definition [1] \( \eta \) must be \( Y \)-just for some \( Y \subseteq \mathcal{M} \). So \( \eta \) can be decomposed into an \( X \)-just path \( \eta_1 \) of \( I_1 \), a \( Z \)-just path \( \eta_2 \) of \( F[f] \) and a \( W \)-just path \( \eta_3 \) of \( I_2 \) for certain \( X, Z, W \subseteq \mathcal{M} \). By assumption, \( c_1 \in X \). Moreover, \( \eta_2 \) can be decomposed into an \( f^{-1}(Z) \)-just path \( \eta_4 \) of \( F \). Since in \( \tilde{F} \) the \( c_1 \) of \( F[f] \) requires synchronisation with the \( c_1 \) of \( I_1 \), and \( \eta_1 \) has only finitely many occurrences of \( c_1 \), it follows that \( \eta_0 \) has only finitely many occurrences of \( c_1 \), and thus that \( \eta_4 \) has only finitely many occurrences of \( r_1 \). Since \( F \) satisfies Requirement [1] saying that the system will always return to a state where it remains ready to accept the next request \( r_1 \) until it arrives, \( r_1 \in f^{-1}(Z) \). Hence \( c_1 \in Z \). By Definition [1] this contradicts the justness of \( \eta \).

   Above we have shown that each occurrence of \( r_1 \) in a just path of \( \tilde{F} \), which corresponds to an occurrence of \( r_1 \) in the subprocess \( I_1 \), is followed by an occurrence of \( c_1 \) in \( I_1 \). This occurrence of \( c_1 \) in \( I_1 \) must be a synchronisation with an occurrence of \( c_1 \) in \( F[f] \), and by Requirement [2] for each occurrence of \( c_1 \) in \( F[f] \) is followed by an occurrence of \( t_1 \) in \( F[f] \), and hence in \( \tilde{F} \).

3. By Requirement [3] for \( F \), on each finite path from \( F[f] \) there are no more occurrences of \( t_1 \) than of \( c_1 \). Moreover, on each finite path from \( I_1 \) there are no more occurrences of \( c_1 \) than of \( r_1 \). Since in \( \tilde{F} \) each occurrence of \( c_1 \) in \( F[f] \) needs to synchronise with an occurrence of \( c_1 \) in \( I_1 \), it follows that on each finite path from \( \tilde{F} \) there are no more occurrences of \( c_1 \) than of \( r_1 \).

4. Requirement [4] holds for \( \tilde{F} \) because it holds for \( F \).

Now assume that \( \tilde{F} \) satisfies Requirements [2][4]

1. Suppose that \( F \) would fail Requirement [1] say for \( i = 1 \). Then it has a \( Z \)-just path with \( r_1 \notin Z \). Therefore \( F[f] \) has an \( f(Z) \)-just path with \( c_1 \notin f(Z) \). This path can be synchronised with a \( c_1 \)-just path of \( I_1 \) into a just path of \( \tilde{F} \) in which an occurrence of \( r_1 \) follows the last occurrence of \( t_1 \), thereby violating Requirement [2] for \( \tilde{F} \).

2. Suppose that \( F \) would fail Requirement [2] say for \( i = 1 \). Then it has a just path with an occurrence of \( r_1 \) past the last occurrence of \( t_1 \). Therefore \( F[f] \) has a \( Z \)-just path with \( r_1 \notin Z \) and an occurrence of \( c_1 \) past the last occurrence of \( t_1 \). This path can be synchronised with a \( r_1 \)-just path of \( I_1 \) into a just path of \( \tilde{F} \) in which an occurrence of \( r_1 \) follows the last occurrence of \( t_1 \), thereby violating Requirement [2] for \( \tilde{F} \).

3. Suppose \( F \) had a finite path with more occurrences of \( t_1 \) than of \( r_1 \), then through synchronisation a finite path of \( \tilde{F} \) could be constructed with more occurrences of \( t_1 \) than of \( r_1 \).

4. Requirement [3] holds for \( F \) because it holds for \( \tilde{F} \).

\( \Box \)
Recall that $G_2$, given by $G_2 \overset{\text{def}}{=} r_1.t_1.e.G_2 + r_2.t_2.e.G_2$, satisfies Requirements 2-4. Converting this $G_2$ to the process $\hat{G}_2$ of the form $(I_1 \mid G \mid I_2) \setminus c_1 \setminus c_2$, as defined above, results in the specification of Section 11 without the additional fairness specification, and hence in the loss of Requirement 2.

14 (In)Correct Correctness Proofs of Peterson’s and Dekker’s Protocols

It is widely accepted that Peterson’s mutual exclusion protocol [51] implements a fair scheduler, and that implementing Peterson’s algorithm in a CCS-like language should be easy. In fact Peterson’s algorithm has been specified in CCS-like languages several times, e.g. [61, 10, 59, 1]. All these papers present essentially the same rendering of Peterson’s algorithm in CCS or some other progress algebra, differing only in insignificant implementation details. This seems to contradict our main result (Theorem 1).

Peterson’s Mutual Exclusion Protocol deals with two concurrent processes A and B that want to alternate critical and noncritical sections. Each of these processes will stay only a finite amount of time in its critical section, although it is allowed to stay forever in its noncritical section. The purpose of the algorithm is to ensure that they are never simultaneously in the critical section, and to guarantee that both processes keep making progress. Pseudocode is depicted in Figure 1.

The processes use three variables. The Boolean variable $\text{ready}_A$ can be written by process A and read by process B, whereas $\text{ready}_B$ can be written by B and read by A. By setting $\text{ready}_A$ to true, process A signals to process B that it wants to enter the critical section. The variable $\text{turn}$ is a shared variable; it can be written and read by both processes. Its use is the brilliant part of the algorithm. Initially $\text{ready}_A$ and $\text{ready}_B$ are both false and $\text{turn} = A$.

Peterson’s algorithm implements a mutual exclusion protocol and hence should satisfy the safety property that at any time only one process accesses the critical system, i.e.

$$G(\neg((\ell_4 \lor \ell_5) \land (m_4 \lor m_5))).$$

Here, $\ell_i$ and $m_j$ refer to line numbers of the pseudocode (Figure 1). As convention we assume that line numbers refer to a state in the execution of the code where the command of the line has already been executed. Most papers, including [10], [4], concentrate on the issue of mutual exclusion only, and that is done correctly in the CCS rendering. When safety properties are considered only, no fairness or
progress assumption is needed: in the worst case some (or all) processes do not progress and hence never enter the critical section—the safety property still holds. As usual, a safety property should therefore be accompanied with a liveness property. In case of Peterson’s protocol such a property is that any process that wants to enter the critical section will at some point reach the critical section. We consider two possibilities to characterise this property:

\[ G(\ell_1 \Rightarrow F(\ell_4)), \quad \text{and} \quad G(\ell_2 \Rightarrow F(\ell_4)) \]

Both properties have the form \( G(r_1 \Rightarrow F(t_1)) \)—the property discussed in this paper—\( \ell_4 \) indicates that the process enters the critical section. Both \( \ell_1 \) and \( \ell_2 \) could play the role of the grant request \( r_1 \). Although it seems surprising, we will show that there is a fundamental difference between these two formulas.

To show how Peterson’s algorithm yields an instance of our fair scheduler, we consider the action \( t_1 \) to be taken when an execution passes through state \( \ell_4 \), thereby interpreting \( t_1 \) as granting access to the critical section. The action \( e \) is taken when the execution passes through state \( \ell_5 \), marking the exit of the critical section. Peterson’s code, in combination with the mutual exclusion property, ensures that Requirement \( \textbf{1} \) of our fair scheduler is satisfied. We consider \( r_1 \) to be taken when an execution passes through state \( \ell_1 \), so that the liveness property \( G(\ell_1 \Rightarrow F(\ell_4)) \) ensures Requirement \( \textbf{2} \). Requirement \( \textbf{1} \) is satisfied, because as soon as the environment of the protocol leaves the noncritical section, thereby getting ready to enter the critical one, the protocol is considered to take the action \( r_1 \). Finally, Requirement \( \textbf{3} \) is obviously ensured by Peterson’s code.

In combination with this insight, our main result (Theorem \( \textbf{1} \)) entails that the rendering of Peterson’s algorithm in CCS found in the literature cannot be correct, as long as the semantics of CCS is fortified with at most justness. To prove liveness of Peterson’s protocol, at least weak fairness is required\( ^\text{14} \). In the literature we found only two papers that investigate liveness properties of this protocol: \( ^\text{61} \) and \( ^\text{59} \). Neither of these papers employs fairness or justness properties.

Walker \( ^\text{61} \) tries to prove the correctness of Peterson’s algorithm by automatic methods. He succeeds for the safety property, but could not establish the liveness property in full generality; however Walker succeeded in proving it when restricting attention to runs in which infinitely many visible actions occur. This appears to be Walker’s method of imposing a progress assumption. Although this is strictly speaking not in contradiction with our results, our proofs trivially extend to the case of considering only runs in which infinitely many visible actions occur. Hence Walker’s result seems to be in contradiction to Theorem \( \textbf{1} \). A detailed analysis reveals that Walker uses line \( \ell_2 \) as request action to indicate interest to enter the critical section. So he shows that \( G(\ell_2 \Rightarrow F(\ell_4)) \). That means that the shared variable \textit{ready\textsubscript{A}} must be set—only then \( \ell_2 \) evaluates to true. His request action is set right after setting this variable. However, following our proof, the reason that the CCS rendering of Peterson does not work, is that it is possible that process A never gets a chance to set the shared variable \textit{ready\textsubscript{A}} to true, because the other process is too busy reading it all the time (even when it enters the critical section between any two reads). So, it is a possible scenario that process A will never execute line \( \ell_2 \), although it wants to enter the critical section.

\( ^\text{13} \) We only give the liveness property for process \( \textit{A} \); the one for process \( \textit{B} \) is similar.

\( ^\text{14} \) Whether weak fairness suffices depends on the interpretation of enabledness (cf. Section \( \textbf{2} \))
In Peterson’s original thinking, process B could not prevent process A from writing by reading a shared variable; but in the CCS model this is quite possible: the read action can only be represented as a transition that is in conflict with the write action; only after this transition is taken does the process return to a state where the write is enabled. So when Walker [61] establishes $G(\ell_2 \Rightarrow F(\ell_4))$ he merely shows that when the first hurdle is taken successfully the process will surely enter the critical section. What he cannot establish is that a process that is ready to enter the critical section will succeed in setting $\text{ready}A$. The correct modelling of Peterson’s liveness property thus places action $r_1$ before setting the variable $\text{ready}A$ to true, i.e.

$$G(\ell_1 \Rightarrow F(\ell_4)).$$

In terms of our description of a fair scheduler, the action $r_1$ of Walker (at position $\ell_2$) does not meet Requirement 1.

The analysis of the work of Walker shows that there is a fine line between correct and incorrect modelling. In fact it looks reasonable to prove $G(\ell_2 \Rightarrow F(\ell_4))$ instead of $G(\ell_1 \Rightarrow F(\ell_4))$. There is no formal way to avoid such mistakes; only careful (informal) reasoning.

Roughly the same modelling, but in which the request $r_1$ is identified with setting the shared variable $\text{ready}A$ to true, occurs in Valmari & Setälä [59]. The consequences are the same.

Dekker’s mutual exclusion protocol [20,21] is another well-known algorithm that implements a fair scheduler. We found two papers in the literature that analyse liveness of this protocol.

Esparza & Burns [22] follow in the footsteps of Walker and prove the correctness of Dekker’s mutual exclusion algorithm in the Box Calculus without postulating a fairness assumption. According to our results, this is impossible as well. Indeed, as in [59], Esparza & Burns model the request $r_1$ to be the action of setting a shared variable, which again violates the property that a process that wants to enter the critical section can always succeed at least in making a request to that effect.

Corradini, Di Berardini & Vogler [14] specify Dekker’s algorithm in the CCS-like process algebra PAFAS. They also prove the correctness of the algorithm. This paper models the relevant liveness properties correctly, as far as we can see, but explicitly makes different assumptions on the driving force that keeps the system running. First they consider a notion of ‘fairness of actions’ that appears to be similar to our justness assumption [15] and they show that their model of Dekker’s protocol fails to have the required liveness property. This result is entirely consistent with ours. In fact we generalise their negative result about the correctness of a particular rendering in PAFAS of a particular protocol for mutual exclusion to a general statement quantifying of all renderings of all such protocols.

Next they consider a stronger notion of fairness called ‘fairness of components’, stemming from [15], and, under this assumption, establish the correctness of the

---

15 It differs in a crucial way, however, namely by treating each action as output. As a consequence, under fairness of actions the process $F_1|F_2$ of Section [11] is guaranteed to perform each of the actions $r_1$ and $r_2$ infinitely often. To model a protocol where the action $r_1$ is not forced to occur, a $\tau$-loop is inserted at each location where $r_1$ is enabled.
algorithm\textsuperscript{14} The present paper augments this result by saying that a fairness notion as strong as ‘fairness of components’ is actually needed.

In Part I we pointed out that our result holds for CCS+justness, CCS+progress and CCS without any progress assumption. However a fair scheduler can be implemented when a fairness assumption is assumed; fairness of components appears to be sufficient.

Part III Proofs

15 Fair Schedulers Cannot be Rendered in Petri Nets—Formalisation

This section introduces Petri nets and rephrases Theorem 10 in terms of Petri nets. We inherit the sets \( \text{Act} \) of actions and \( \mathcal{H} \subseteq \text{Act} \) of handshaking communications from Section 8, and the set \( \emptyset \) of output actions from Section 9.

A multiset over a set \( X \) is a function \( A : X \to \mathbb{N} \), i.e. \( A \in \mathbb{N}^X \). The function \( \emptyset : X \to \mathbb{N} \), given by \( \emptyset(x) := 0 \) for all \( x \in X \), is the empty multiset over \( X \).

For multisets \( A \) and \( B \) over \( X \) we write \( A \leq B \) iff \( A(x) \leq B(x) \) for all \( x \in X \);

\( A \cap B \) denotes the multiset over \( X \) with \( (A \cap B)(x) := \min(A(x), B(x)) \),

\( A + B \) denotes the multiset over \( X \) with \( (A + B)(x) := A(x) + B(x) \), and

\( A - B \) is only defined if \( B \leq A \) and then denotes the multiset over \( X \) with \( (A - B)(x) := A(x) - B(x) \).

A multiset \( A \) with \( A(x) \leq 1 \) for all \( x \) is identified with the (plain) set \( \{ x \mid A(x) = 1 \} \).

Definition 2 A (labelled) Petri net (over \( \text{Act} \)) is a tuple \( N = (S, T, F, M_0, \ell) \) with

- \( S \) and \( T \) disjoint sets (of places and transitions),
- \( F : ((S \times T) \cup (T \times S)) \to \mathbb{N} \) (the flow relation including arc weights),
- \( M_0 : S \to \mathbb{N} \) (the initial marking), and
- \( \ell: T \to \text{Act} \) (the labelling function).

When a Petri net represents a concurrent system, a global state of this system is given as a marking, a multiset of places. The initial state is \( M_0 \).

The behaviour of a Petri net is defined by the possible moves between markings \( M \) and \( M' \), which take place when a transition \( u \) fires. In that case, \( u \) consumes \( F(s,u) \) tokens from each place \( s \). Naturally, this can happen only if \( M \) makes all these tokens available in the first place. Moreover, \( u \) produces \( F(u,s) \) tokens in each \( s \).

Definition 3 Let \( N = (S,T,F,M_0,\ell) \) be a Petri net and \( u \in T \). The multisets \( \cdot u, \cdot u^*: S \to \mathbb{N} \) are given by \( \cdot u(s) = F(s,u) \) and \( \cdot u^*(s) = F(u,s) \) for all \( s \in S \).\textsuperscript{17}

\textsuperscript{14} Fairness of components is a form of weak fairness, requiring that if a component from some point onwards is enabled in each state, an action from that component will eventually be scheduled. Here a component is enabled if an action from that component is enabled, possibly in synchronisation with an action from outside that component. Under this notion of fairness, the system \( E \) from Section 8, defined by \( E \overset{\text{def}}{=} a.E + b.0 \), is not ensured to do a \( b \) eventually. However, the composition \( (E[b.c.0]\backslash b) \) is ensured to do a \( c \) eventually, because the component \( b.c.0 \) is enabled in every state.

\textsuperscript{17} Here, we slightly deviate from standard notation \([57]\), where \( \cdot u \) and \( \cdot u^* \) are usually plain sets, obtained from our multisets by abstracting from the multiplicities of their elements. We prefer to retain this information, so as to shorten various formulas.
elements of \( \cdot u \) and \( u' \) are called pre- and postplaces of \( u \), respectively. Transition \( u \in T \) is enabled from the marking \( M \in \mathbb{N}^S \)—notation \( M[\cdot u] \)—if \( \cdot u \leq M \). In that case firing \( u \) yields the marking \( M' := M - \cdot u + u' \)—notation \( M[\cdot u]M' \).

A path \( \pi \) of a Petri net \( N \) is an alternating sequence \( M_0u_1M_1u_2M_2u_3 \ldots \) of markings and transitions, starting from the initial marking \( M_0 \) and either being infinite or ending in a marking \( M_n \), such that \( M_k[\cdot u_k]M_{k+1} \) for all \( k \leq n \). An action \( \alpha \in \text{Act} \) occurs on a path \( \pi \) if there is a transition \( u_i \) with \( \ell(u_i) = \alpha \). A marking is reachable if it occurs in such a path. The Petri net \( N \) is safe if all reachable markings \( M \) are plain sets, meaning that \( M(s) \leq 1 \) for all places \( s \). It is a structural conflict net \cite{conflict} if \( \cdot u + \cdot v \leq M \Rightarrow \cdot u \cap \cdot v = \emptyset \) for all reachable markings \( M \) and all transitions \( u \) and \( v \). Note that any safe Petri net is a structural conflict net. In this paper we restrict attention to structural conflict nets with the additional assumptions that \( \cdot u = \emptyset \) for no transition \( u \), and that all reachable markings are finite. In the remainder we refer to these structures as nets. For the purpose of establishing Theorem \cite{Wim} we could just as well have further restricted attention to safe Petri nets whose reachable markings are finite.

On a path \( \pi = M_0u_1M_1u_2M_2u_3 \ldots \) a transition \( v \) is continuously enabled from position \( k \) onwards if \( M_k[\cdot v] \) and \( \cdot v \cap \cdot u_i = \emptyset \) for all \( i > k \). This implies that \( \cdot v \leq M_i \) for all \( i \geq k \). If such a transition \( v \) exists we say that \( \pi \) is \( \ell(v) \)-enabled. A path is just or complete if it is \( o \)-enabled for no non-blocking action \( o \in \sigma \cup \{ \tau \} \).

Now we have all the necessary definitions to state that our fair scheduler cannot be realised as a net.

**Theorem 3** There does not exist a net \( N \) such that:

1. any complete path of \( N \) that has finitely many occurrences of \( r_i \) is \( r_i \)-enabled;
2. on each complete (just) path of \( N \), each \( r_i \) is followed by a \( t_i \);
3. on each finite path of \( N \) there are no more occurrences of \( t_i \) than of \( r_i \); and
4. between each two occurrences of \( t_i \) and \( t_j \) \( i, j \in \{1, 2\} \) an action \( e \) occurs.

In the proof of this theorem we do not use the restriction that \( N \) is a structural conflict net. However, for general Petri nets our definition of a transition that from some points onwards is continuously enabled is not convincing. A better definition would replace the requirement \( \cdot v \cap \cdot u_i = \emptyset \) for all \( i > k \) by \( \cdot v + \cdot u_{i+1} \leq M_i \) for all \( i \geq k \). On structural conflict nets the two definitions are equivalent. On general Petri nets with finite reachable markings and \( \forall u. \cdot u \neq \emptyset \) Theorem \cite{conflict} still holds when employing our earlier definition of being continuously enabled, but that definition arguably leads to Requirement \cite{Wim} being an overly restrictive formalisation of the first requirement of Section \cite{mutex}.

In \cite{conflict} and in \cite{mutex} *mutex problems* are presented that cannot be solved in terms of Petri nets. These results are almost equivalent to Theorem \cite{conflict} but, as discussed in the Section \cite{mutex} lack the generality needed to infer Theorem \cite{Wim} from Theorem \cite{conflict}.

16 Fair Schedulers Cannot be Rendered in Petri Nets—Proof

In this section we suppose that there exists a net \( N \) meeting the requirements of Theorem \cite{conflict}. We establish various results about this hypothetical net \( N \), ultimately leading to a contradiction. This will constitute the proof of Theorem \cite{conflict}.
16.1 Embellishing Paths into Complete Paths

A firing sequence of $N$ is a sequence $\sigma = u_1 u_2 u_3 \ldots$ of transitions such that there exists a path $\pi = M_0 u_1 M_1 u_2 M_2 u_3 \ldots$ of $N$. Note that $\pi$ is uniquely determined by $\sigma$ (and $M_0$); we call it $\text{ph}(\sigma)$. Likewise, $\sigma$ is determined by $\pi$, and we call it $\text{FS}(\pi)$.

A firing sequence $\sigma' = v_1 v_2 v_3 \ldots$ embellishes a firing sequence $\sigma = u_1 u_2 u_3 \ldots$ iff $\sigma'$ can be obtained out of $\sigma$ through insertion of non-blocking transitions; that is, if there exists a monotone increasing function $f : \mathbb{N} \to \mathbb{N}$—thus satisfying $i < j \Rightarrow f(i) < f(j)$—with $v_f(i) = u_i$ for all $i > 0$ and $\ell(v_i) \in \mathcal{O} \cup \{v\}$ for any index $j$ not of the form $f(i)$. A path $\pi'$ embellishes a path $\pi$ iff $\text{FS}(\pi')$ embellishes $\text{FS}(\pi)$.

Given a firing sequence $\sigma = u_1 u_2 \ldots$ of length $\geq k$ and a transition $w$, let $\sigma \oplus_k w$ denote the sequence $u_1 u_2 \ldots u_k w u_{k+1} u_{k+2} \ldots$ obtained by inserting $w$ in $\sigma$ at position $k$.

**Lemma 1** Let $\sigma$ be a firing sequence of length $\geq k$ and $w$ a transition that on $\text{ph}(\sigma)$ is continuously enabled from position $k$ onwards. Then $\sigma \oplus_k w$ is a firing sequence.

**Proof** Let $\text{ph}(\sigma) = M_0 u_1 M_1 u_2 M_2 u_3 \ldots$. Define $M'_i := M_i - *w + w*$ for $i \geq k$. Then $M_0 u_1 M_1 u_2 \ldots u_k M_k w M'_k u_{k+1} M'_k u_{k+2} \ldots$ is again a path of $N$, using that $*w \leq M_i$ and $u_{i+1} \leq M_i - *w$ for all $i \geq k$. □

If $\pi$ is a path and $w$ a transition that on $\pi$ is continuously enabled from position $k$ onwards, then $\pi \oplus_k w$ abbreviates $\text{ph}(\text{FS}(\pi) \oplus_k w)$.

A path $\pi = M_0 u_1 M_1 u_2 M_2 u_3 \ldots$ of $N$ is $(k, n)$-incomplete if $k$ is the smallest number such that there is a transition $w$ with $\ell(w) \in \mathcal{O} \cup \{v\}$—called a witness of the $(k, n)$-incompleteness of $\pi$—that is continuously enabled from position $k$ onwards, and $n$ is the number of places $s$ of $N$ such that $s \in *w$ for a witness $w$ of the $(k, n)$-incompleteness of $\pi$. Since the reachable marking $M_k$ is always finite, so are the numbers $n$. Note that a path is $(k, n)$-incomplete for some finite $k$ and $n$ iff it is not complete; henceforth we call a complete path $(\infty, 0)$-incomplete. If a path $\pi$ is $(k, n)$-incomplete, and a path $\rho$ is $(h, m)$-incomplete, then we call $\rho$ less incomplete than $\pi$ if $\rho$ has the same prefix up to position $k$ as $\pi$ and either $h < k$ or $h = k \land m < n$.

**Lemma 2** Let $i \geq 0$, $\pi$ be a $(k, n)$-incomplete path of the net $N$ with at least $k+i$ transitions, and $w$ a witness of the $(k, n)$-incompleteness of $\pi$. Then $\pi \oplus_{k+i} w$ is less incomplete than $\pi$.

**Proof** Suppose $\pi \oplus_{k+i} w$ is $(h, m)$-incomplete with $h \leq k$, and let $v$ be a witness of the $(h, m)$-incompleteness of $\pi \oplus_{k+i} w$. Then $v \oplus_{k+i} w$ is continuously enabled from position $h$ onwards. Let $M_h$ be the marking occurring at position $h$ in $\pi$, or equivalently in $\pi \oplus_{k+i} w$. Then $v' \leq M_h$ and $v' \cap u = \emptyset$ for all transitions $u$ occurring in $\pi \oplus_{k+i} w$ past position $h$. This includes all transitions $u$ occurring in $\pi$ past position $h$, so $v$ is continuously enabled from position $h$ onwards also on $\pi$. It follows that $h = k$ and any witness of the $(k, m)$-incompleteness of $\pi \oplus_{k+i} w$ is also a witness of the $\pi$-incompleteness of $\pi$. Moreover, $v' \cap *w = \emptyset$, and since $*w \neq \emptyset$ this implies $m < n$. □

**Lemma 3** Any infinite path in $N$ is embellished by a complete path.
Proof Let \( \pi \) be the given path. We build a sequence \( \pi_i \) of paths in \( N \) that all embellish \( \pi \), such that, for all \( i \), \( \pi_{i+1} \) is less incomplete than \( \pi_i \) and the first \( 2i \) transitions of \( \pi_i \) and \( \pi_{i+1} \) are the same.

We start by taking \( \pi_0 \) to be \( \pi \). If at any point we hit a path \( \pi_i \) that is complete, our work is done. Otherwise, given the \((k, n)\)-incomplete path \( \pi_i \), for some \( k \) and \( n \), pick a witness \( w \) of the \((k, n)\)-incompleteness of \( \pi_i \) and take \( \pi_{i+1} := \pi_i \oplus_{k+2i} w \). This path exists by Lemma 1, since \( w \) is continuously enabled from position \( k \) onwards, and hence also from position \( k + 2i \) onwards. By construction \( \pi_{i+1} \) embellishes \( \pi_i \) and hence \( \pi \). By Lemma 2 \( \pi_{i+1} \) is less incomplete than \( \pi_i \). Moreover, the first \( 2i \) transitions of \( \pi_i \) and \( \pi_{i+1} \) are the same.

If at no point we hit a path \( \pi_i \) that is complete, let \( \rho := \lim_{i \to \infty} \pi_i \). This limit clearly exists: for any \( i \in \mathbb{N} \) the first \( 2i \) transitions of \( \rho \) are the first \( 2i \) transitions of \( \pi_i \) (and thus also of \( \pi_j \) for any \( j > i \)). We show that \( \rho \) is complete and embellishes \( \pi \).

For the latter property, the \( i \)th transition \( u_i \) of \( \pi \) must also occur in \( \pi_i \), and no further than at position \( 2i \), for \( \pi_i \) is an embellishment of \( \pi \) obtained by adding only \( i \) transitions. As in the sequence \((\pi_j)_{j=0}^{\infty} \) past index \( i \) no further changes occur in the first \( 2i \) transitions, the transition \( u_i \) also occurs in \( \rho \). Given the construction, this implies that \( \rho \) embellishes \( \pi \). The same argument shows that \( \rho \) embellishes \( \pi_i \) for each \( i \in \mathbb{N} \).

Now suppose that \( \rho \) is incomplete. Then there is a non-blocking transition \( w \) that on \( \rho \), from some position \( k \) onwards, is continuously enabled. Let \( i \in \mathbb{N} \) be an index such that \( \pi_i \) is \((k, n)\)-incomplete for some \( k > h \). Such an \( i \) must exist, as the members of \((\pi_i)_{i=0}^{\infty} \) become less incomplete with increasing \( i \). Let \( M_h \) be the marking occurring at position \( h \) in \( \rho \). Then \( M_h \) also occurs at position \( h \) in \( \pi_i \), as the first \( k + 2i \) transitions of \( \pi_i \) are the same for all \( \pi_j \) with \( j \geq i \), and thus for \( \rho \). Now \( \bullet w \leq M_h \) and \( \bullet w \cap \bullet u = \emptyset \) for all transitions \( u \) occurring in \( \rho \) past position \( h \). Since \( \rho \) embellishes \( \pi_i \), this includes all transitions \( u \) occurring in \( \pi_i \) past position \( h \), so \( w \) is continuously enabled from position \( h \) onwards also on \( \pi_i \), contradicting the \((k, n)\)-incompleteness of \( \pi_i \).

\( \square \)

Lemma 4 Any finite path in \( N \) can be extended to a complete path in \( N \), such that all transitions in the extension have labels in \( \emptyset \cup \{ \tau \} \).

Proof This is a simpler variant of the previous proof. Let \( \pi \) be the given path. We build a sequence \( \pi_i \) of paths in \( N \) that all extend \( \pi \), such that, for all \( i \), \( \pi_{i+1} \) is less incomplete than \( \pi_i \) and extends \( \pi_i \) by one transition.

We start by taking \( \pi_0 \) to be \( \pi \). If at any point we hit a path \( \pi_i \) that is complete, our work is done. Otherwise, given the \((k, n)\)-incomplete path \( \pi_i \), for some \( k \) and \( n \), pick a witness \( w \) of the \((k, n)\)-incompleteness of \( \pi_i \) and obtain \( \pi_{i+1} := \text{PH}(\text{fs}(\pi_i)w) \) by appending transition \( w \) to \( \pi_i \). By construction \( \pi_{i+1} \) extends \( \pi_i \) by one non-blocking transition and hence extends \( \pi \). By Lemma 2 \( \pi_{i+1} \) is less incomplete than \( \pi_i \).

If at no point we hit a path \( \pi_i \) that is complete, let \( \rho := \lim_{i \to \infty} \pi_i \). Clearly, \( \rho \) extends \( \pi \). That \( \rho \) is complete follows exactly as in the previous proof. \( \square \)

16.2 Paths of the Hypothetical Fair Scheduler

Lemma 5 Our hypothetical net \( N \) has a path with no occurrences of (transitions labelled) \( r_1 \), but infinitely many occurrences of \( r_2 \).
Lemma 6 \( N \) has a path with exactly one occurrence of \( r_1 \), none of \( t_1 \), and infinitely many occurrences of \( t_2 \).

Proof Let \( \pi \) be the path found by Lemma 6. By Lemma 3 this path is embellished by a complete path \( \pi' \), that thus has no occurrences of \( r_1 \) and infinitely many of \( r_2 \). By Requirement 2 \( \pi' \) has infinitely many occurrences of \( t_2 \), and by Requirement 3 it has no occurrences of \( t_1 \). By Requirement 4 \( \pi' \) is \( r_1 \)-enabled. Let \( w \) be a transition labelled \( r_1 \) that is on \( \pi \) is continuously enabled from position \( k \) onwards. By Lemma 4 \( N \) has a path \( \pi \oplus_k w \), obtained from \( \pi' \) by inserting transition \( w \) in position \( k \). That path has exactly one occurrence of \( r_1 \), none of \( t_1 \), and infinitely many of \( t_2 \).

Lemma 7 \( N \) has a \( t_1 \)-enabled path with infinitely many occurrences of \( t_2 \).

Proof Let \( \pi \) be the path found by Lemma 6. We build a sequence \( \pi_i \) of paths in \( N \) that all embellish \( \pi \) and do not contain \( t_1 \). For all \( i \), \( \pi_{i+1} \) is incomplete than \( \pi_i \) and the first \( 2i \) transitions of \( \pi_i \) and \( \pi_{i+1} \) are the same. Since the \( \pi_i \) embellish \( \pi \), they have exactly one occurrence of \( r_1 \), and infinitely many of \( t_2 \). Moreover, by Requirement 4 none of the \( \pi_i \) can be complete.

We start by taking \( \pi_0 \) to be \( \pi \). If at any point we hit a path \( \pi_i \) that is \( t_1 \)-enabled, our work is done. Otherwise, given the \(( k, n )\)-incomplete path \( \pi_i \), for some \( k \) and \( n \), pick a witness \( w \) of the \(( k, n )\)-incompleteness of \( \pi_i \) and take \( \pi_{i+1} := \pi_i \oplus_{k+2i} w \). This path exists by Lemma 1 since \( t \) is continuously enabled from position \( k \) onwards, and hence also from position \( k + 2i \) onwards. Note that \( \ell ( w ) \neq t_1 \), since \( \pi_i \) is not \( t_1 \)-enabled. Hence \( \pi_{i+1} \) does not contain \( t_1 \). By construction \( \pi_{i+1} \) embellishes \( \pi_i \) and hence \( \pi \). By Lemma 2 \( \pi_{i+1} \) is less incomplete than \( \pi_i \). Moreover, the first \( 2i \) transitions of \( \pi_i \) and \( \pi_{i+1} \) are the same.

If at no point we hit a path \( \pi_i \) that is \( t_1 \)-enabled, let \( \rho := \lim_{i \to \infty} \pi_i \). Exactly as in the proof of Lemma 3 it follows that \( \rho \) is complete and embellishes \( \pi \). Since \( t_1 \) does not occur on any of the \( \pi_i \), it does not occur on \( \rho \). However, \( r_1 \) does occur on \( \rho \), since it occurred on \( \pi \). This is in contradiction with Requirement 2. Therefore, the assumption that at no point we hit a path \( \pi_i \) that is \( t_1 \)-enabled must be wrong.

Proof of Theorem 3 Let \( \pi \) be the path found in Lemma 1. It must have a finite prefix \( \pi' \) ending with an occurrence of \( t_2 \), such that a transition \( w \) labelled \( t_1 \) is enabled at the last state of \( \pi' \). Extending \( \pi' \) with \( w \) yields a finite path of \( N \) violating Requirement 4.
17 An Operational Petri Net Semantics of CCS

This section presents an operational Petri net semantics of CCS\textsuperscript{1}, following Degano, De Nicola & Montanari [19]. It associates a Petri net \([P]\) with each CCS\textsuperscript{1} expression \(P\). We establish that this Petri net is safe, all its reachable markings are finite, and there are no transitions \(u\) with \(^*u = \emptyset\); hence it is one of the nets considered in Section 15. In Section 18 we will show that if a CCS\textsuperscript{1} expression \(P\) enriched with an initial state—for \(P\) requirements of Theorem 1 then the Petri net \([F]\) by appointing \(P\) transitions not reachable from \(P\) and there are no transitions \(u\). In the same vein, an operational Petri net semantics yields one big labelled transition system for the entire language. Each individual CCS\textsuperscript{1} expression \(P\) appears as a state in this LTS. If desired, a process graph—an LTS enriched with an initial state—for \(P\) can be extracted from this system-wide LTS by appointing \(P\) as the initial state, and optionally deleting all states and transitions not reachable from \(P\). In the same vein, an operational Petri net semantics yields one big Petri net for the entire language, but without an initial marking. We call such a Petri net unmarked. Each process \(P \in \text{T}_{\text{CCS}}\) corresponds to a marking \(\text{dec}(P)\) of that net. If desired, a Petri net for \(P\) can be extracted from this system-wide net by appointing \(\text{dec}(P)\) as its initial marking, and optionally deleting all places and transitions not reachable from \(\text{dec}(P)\).

The set \(G_{\text{CCS}}\) of places in the net—the grapes of [10]—is the smallest set including:

\[
\begin{align*}
\alpha.P & \text{ prefixing} \quad \sum_{i \in I} P_i & \text{ agent identifier} \\
\mu & \text{ left parallel component} \quad |\mu & \text{ right component} \\
& \text{ choice} & \mu[a] \text{ restriction} \\
\end{align*}
\]

for \(A \in \mathcal{A}, \alpha \in \text{Act}, P_i \in \text{T}_{\text{CCS}}, a \in \mathcal{H}, \mu \in \text{G}_{\text{CCS}}\), index sets \(I\), and relabellings \(f\). The mapping \(\text{dec} : \text{T}_{\text{CCS}} \rightarrow \mathcal{P}(\text{G}_{\text{CCS}})\) decomposing a process expression into a set of grapes is inductively defined by:

\[
\begin{align*}
\text{dec}(\alpha.P) &= \{\alpha.P\} & \text{dec}(A) &= \{A\} \\
\text{dec}(\sum_{i \in I} P_i) &= \{\sum_{i \in I} P_i\} & \text{dec}(P\{Q\}) &= \text{dec}(P) \cup \text{dec}(Q) \\
\text{dec}(P\{a\}) &= \text{dec}(P)\{a\} & \text{dec}(P\{f\}) &= \text{dec}(P)\{f\}
\end{align*}
\]

Here \(H[f], H\{a\}, H\) and \(|H|\) are understood element by element; e.g. \(H[f] = \{\mu[f] \mid \mu \in H\}\). Moreover the binding is important, meaning that \(|(H)| \neq |(H)|\).

We construct the unmarked Petri net \((S, T, F, \ell)\) of CCS\textsuperscript{1} with \(S := \text{G}_{\text{CCS}}\), specifying the triple \((T, F, \ell)\) as a ternary relation \(\rightarrow \subseteq \mathbb{N}^S \times \text{Act} \times \mathbb{N}^S\). An element \(H, J, \alpha.\) of this relation denotes a transition \(u \in T\) with \(\ell(u) = \alpha\) such that \(u = H\) and \(u* = J\). The transitions \(H \rightarrow J\) are derived from the rules of Table 2.

Henceforth, we write \(M[\alpha]\) \(M'\), for markings \(M, M' \in \mathbb{N}^{G_{\text{CCS}}}\) and \(\alpha \in \text{Act}\), if there exists a transition \(u\) with \(M[u]M'\) and \(\ell(u) = \alpha\). In that case \(M = H + K\) and \(M' = J + K\) for multisets of places \(H, J, K : \text{G}_{\text{CCS}} \rightarrow \mathbb{N}\) with \(H \rightarrow J\).

The following theorem says that function \(\text{dec}\) is a strong bisimulation ([15]) between the LTS and the unmarked Petri net of CCS\textsuperscript{1}. Since markings of the form \(\text{dec}(R)\) are plain sets (rather than multisets), it also follows that the Petri net of each CCS\textsuperscript{1} expression is safe.

**Theorem 4** If \(R \rightarrow R'\) for \(R, R' \in \text{T}_{\text{CCS}}\) and \(\alpha \in \text{Act}\) then \(\text{dec}(R)[\alpha]\) \(\text{dec}(R')\). Moreover, if \(\text{dec}(R)[\alpha]\) \(M\) then there is a \(R' \in \text{T}_{\text{CCS}}\) with \(R \rightarrow R'\) and \(\text{dec}(R') = M\).
Table 2 Operational Petri net semantics of CCS

\[
\begin{array}{|c|c|c|}
\hline
\{\alpha.P\} \xrightarrow{\alpha} dec(P) & \frac{(dec(P) - K) \xrightarrow{\alpha} J}{\{\sum_{i \in I} P_i\} \xrightarrow{\alpha} J + K} & (j \in I, K \leq dec(P_j)) \\
H \xrightarrow{\alpha} J & \frac{H \xrightarrow{\alpha} J}{H + |K| \xrightarrow{\tau} J + |L|} & \frac{H \xrightarrow{\alpha} J}{H | \xrightarrow{\beta} J} \\
\hline
\end{array}
\]

Proof The first statement follows by induction on the derivability of the transition \(R \xrightarrow{\alpha} R'\) from the rules of Table 1. We only spell out two representative cases; the others are similar or straightforward.

- Suppose \(P|Q \xrightarrow{\alpha} P'|Q\) because \(P \xrightarrow{\alpha} P'\). By induction \(dec(P) \xrightarrow{\alpha} dec(P')\).
  Hence \(dec(P) = H + K\) and \(dec(P') = J + K\) for (multi)sets \(H, J, K \subseteq G_{CCS}\) with \(H \xrightarrow{\alpha} J\). By Table 2 we obtain \(H | \xrightarrow{\beta} J\).
- Suppose \(\alpha \in A\) and \(A \xrightarrow{\alpha} A'\) since \(P \xrightarrow{\alpha} P'\). By induction \(dec(P) \xrightarrow{\alpha} dec(P')\).
  Hence \(dec(P) = H + K\) and \(dec(P') = J + K\) for sets \(H, J, K \subseteq G_{CCS}\) with \(dec(P) - K = H \xrightarrow{\alpha} J\). By Table 2, \(dec(A) = \{A\} \xrightarrow{\alpha} J + K\).

The second statement can be reformulated as

- if \((dec(R) - K) \xrightarrow{\alpha} J\) with \(K \leq dec(R)\),
- then there is a \(R' \in T_{CCS}\) with \(R \xrightarrow{\alpha} R'\) and \(dec(R') = J + K\).

for \(R \in T_{CCS}\) and \(K, J : G_{CCS} \rightarrow \mathbb{N}\). We prove it by induction on the derivability of the transition \(dec(P) - K \xrightarrow{\alpha} J\) from the rules of Table 2.

- Suppose \(dec(R) - K = \{\alpha.P\} \xrightarrow{\alpha} dec(P)\).
  Since the only set \(dec(R)\) containing \(\alpha.P\) is \(\{\alpha.P\}\), we have \(K = \emptyset, J = dec(P)\) and \(R = \alpha.P\). Take \(R' := P\).
- Suppose \(dec(R) - K' = H[f] \xrightarrow{\alpha} J[f]\) because \(H \xrightarrow{\alpha} J\). Then \(R\) must have the form \(P[f]\), so that \(dec(R) = dec(P)[f]\), and \(K'\) must have the form \(K[f]\).
  Thus \(dec(P) - K = H \xrightarrow{\alpha} J\), and by induction there is a \(P' \in T_{CCS}\) with \(P \xrightarrow{\alpha} P'\) and \(dec(P') = J + K\). By Table 1, \(R = P[f] \xrightarrow{\alpha} P'[f]\).
  Moreover, \(dec(P'[f]) = dec(P'[f]) = J[f] + K[f] = J[f] + K'\).
- The case for restriction proceeds likewise.
- Suppose \(dec(R) - K' = H[f] \xrightarrow{\alpha} J[f]\) because \(H \xrightarrow{\alpha} J\). Then \(R\) must have the form \(P|Q\), and \(K' = (dec(P)| - H) + |dec(Q) = K| + |dec(Q)|\), where \(K := dec(P) - H\). Thus \(dec(P) - K = H \xrightarrow{\alpha} J\), so by induction there is a \(P' \in T_{CCS}\) with \(P \xrightarrow{\alpha} P'\) and \(dec(P') = J + K\). By Table 1, \(R = P|Q \xrightarrow{\alpha} P'|Q\).
  Moreover, \(dec(P'|Q) = dec(P') + |dec(Q) = J| + K + |dec(Q) = J| + K'\).
Suppose \( \text{dec}(R) - K' = H|+|K \longrightarrow J|+|L \) because \( H \longrightarrow J \) and \( K \longrightarrow L \). Then \( R \) has the form \( P|Q, \) and \( K' = (\text{dec}(P) - H) + (\text{dec}(Q) - J) = K_1|+|K_2, \) where \( K_1 := \text{dec}(P) - H \) and \( K_2 := \text{dec}(Q) - J. \) Thus \( \text{dec}(P) - K_1 = H \longrightarrow J \) and \( \text{dec}(Q) - K_2 = J \longrightarrow L, \) so by induction there are \( P', Q' \in T_{\text{CCS}} \) with \( P \longrightarrow P', \) \( \text{dec}(P') = J + K_1, \) \( Q \longrightarrow Q' \) and \( \text{dec}(Q') = L + K_2. \) By Table 4 \( R = P|Q \longrightarrow P'|Q'. \) Moreover, \( \text{dec}(P'|Q') = \text{dec}(P')|+|\text{dec}(Q') = J + K_1 + L + |K_2 = J|+|L + K'. \)

The case for the last rule for parallel composition follows by symmetry.

Suppose \( \text{dec}(R) - K' = \{ \sum_{i \in I} P_i \} \longrightarrow J + K \) because \( \text{dec}(P_j) - K \longrightarrow J \) for some \( j \in I. \) Since the only set \( \text{dec}(R) \) containing \( \sum_{i \in I} P_i \) is \( \{ \sum_{i \in I} P_i \}, \) we have \( K' = \emptyset \) and \( R = \sum_{i \in I} P_i. \) By induction, there is a \( P'_j \in T_{\text{CCS}} \) with \( P_j \longrightarrow P'_j \) and \( \text{dec}(P'_j) = J + K. \) By Table 4 \( R = \sum_{i \in I} P_i \longrightarrow P'_j. \)

The case for recursion (agent identifiers) goes likewise. \( \square \)

A trivial induction shows that there are no transitions without preplaces. The following lemma implies that all reachable markings are finite, so that the Petri nets of CCS expressions have all the properties of nets imposed in Section 15.

**Lemma 8** For any \( P \in T_{\text{CCS}} \) the set \( \text{dec}(P) \) is finite.

**Proof** A straightforward induction. \( \square \)

The above operational Petri net semantics of CCS has the disadvantage that initial concurrency in expressions of the form \( \sum_{i \in I} P_i \) or \( A \) is not represented [19]. Although this Petri net semantics matches the LTS semantics of CCS up to strong (interleaving) bisimilarity—and thereby also the standard denotational Petri net semantics of CCS-like operators [30]—, it does not match the standard denotational Petri net semantics up to semantic equivalences that take concurrency explicitly into account. For this reason Olderog [19] provides an alternative operational Petri net semantics that is more accurate in this sense. However, the work of Olderog does not generalise in a straightforward way to the infinite sum construct of CCS, and to unguarded recursion. In fact, a safe Petri net that accurately models the concurrent behaviour of the CCS process \( \sum_{i \in I}(a_i,b)|0 \) would need an uncountable initial marking, and hence falls outside the class of nets we handle in Section 15. Since the accurate modelling of concurrency is not essential for this paper, we therefore use the semantics of [19].

### 18 Fair Schedulers Cannot be Rendered in CCS

**Lemma 9** The mapping \( \text{dec} : T_{\text{CCS}} \to \mathcal{P}(G_{\text{CCS}}) \) is injective.

**Proof** A straightforward induction on the structure of the elements in \( T_{\text{CCS}}. \) \( \square \)

**Lemma 10** Let \( P \in T_{\text{CCS}}. \) For any path \( \pi = \text{dec}(P)u_1M_1u_2M_2u_3 \ldots \) in the unmarked Petri net of CCS there is a unique path \( \bar{\pi} = P \overset{\alpha_1}{\longrightarrow} P_1 \overset{\alpha_2}{\longrightarrow} P_2 \overset{\alpha_3}{\longrightarrow} \ldots \) of the same (finite or infinite) length in the LTS of CCS with \( \text{dec}(P_i) = M_i \) and \( \ell(u_i) = \alpha_i, \) for all \( i. \)

**Proof** A straightforward induction on \( i, \) using Theorem 4 and Lemma 9. \( \square \)
The following observations are based directly on Table 2 and the definition of $dec$.

**Observation 1** Let $dec(P|Q)|w$. Then $M$ has the form $dec(P'|Q')$ and either

- $Q = Q'$ and $dec(P) [v]$ $dec(P)'$ for a $v \in T$ with $\ell(v) = \ell(u)$, $\pi u = \pi v$ and $\pi u = \pi v^*$,
- $P = P'$ and $dec(Q)$ $dec(Q)'$ for a $w \in T$ with $\ell(u) = \ell(u)$, $\pi u = \pi w$ and $\pi u = \pi w^*$,
- or $dec(P) [v]$ $dec(P)'$ and $dec(Q) [w]$ $dec(Q)'$ for $v, w \in T$ with $\ell(v) = c \in \mathcal{H}$,

For each such transition $u$, the transitions $v$ and $w$ discovered above are called the left- and right-projections of $u$, respectively, when they exist. Hence any path $\pi$ starting from a marking $dec(P|Q)$ can be uniquely decomposed into a path $\pi_1$ starting from $dec(P)$ and a path $\pi_2$ starting from $dec(Q)$, notation $\pi \Rightarrow \pi_1|\pi_2$. The path $\pi_1$ fires all existing left-projections of the transitions in $\pi$, in order, and $\pi_2$ its right-projections.

**Lemma 11** If $\pi \Rightarrow \pi_1|\pi_2$ and $\pi_1$ is $\alpha$-enabled, then so is $\pi$.

**Proof** Let $\pi_1 = M_0|u_1|M_1|u_2|M_2|u_3|\ldots$ be $\alpha$-enabled. Then there is a $h \geq 0$ and a transition $v$ with $\ell(v) = \alpha$ such that $M_h[v]$ and $\pi v \cap v_i = \emptyset$ for all $i > h$. Let $\pi = M_0|u_1|M_1|u_2|M_2|u_3|\ldots$. Let $k \geq h$ be such that $v_0 \ldots v_k$ is the sequence of existing left-projections of $u_1 \ldots u_k$. The marking $M_k$ has the form $dec(P_k|Q_k) = dec(P_k)|v|dec(Q_k)$ with $M_k = dec(P_k)$. Since $\pi v \subseteq dec(P_k)$, by Table 2 there exists a transition $v'$ with $\ell(v') = \alpha$ and $\pi v' \subseteq v'$ such that $\pi v' \subseteq dec(P_k)$ and $\pi v' \cap v_k = \emptyset$ for all $i > k$. Moreover, since $\pi v \cap \pi u_i = \emptyset$ for all $i > h$ we have $\pi v' \cap \pi u_i = \emptyset$ for all $i > k$. It follows that $\pi$ is $\alpha$-enabled.

**Lemma 12** If $\pi \Rightarrow \pi_1|\pi_2$, $\pi_1$ is $c$-enabled and $\pi_2$ is $\bar{c}$-enabled, for some $c \in \mathcal{H}$, then $\pi$ is $\tau$-enabled.

**Proof** Let $\pi = M_0|u_1|M_1|u_2|M_2|u_3|\ldots$. Since $\pi \Rightarrow \pi_1|\pi_2$, each marking $M_i$ has the form $dec(P_i|Q_i) = dec(P_i)|v|dec(Q_i)$. By the same reasoning as in the previous proof, there is a $k_1 \geq 0$ and a transition $v$ with $\ell(v) = c$ such that $\pi v \subseteq dec(P_i)$ and $\pi v \cap \pi u_i = \emptyset$ for all $i \geq k_1$. Likewise, there is a $k_2 \geq 0$ and a transition $w$ with $\ell(w) = \bar{c}$ such that $\pi w \subseteq dec(Q_i)$ and $\pi w \cap \pi u_i = \emptyset$ for all $i \geq k_2$.

Let $k = \max(k_1, k_2)$. By the fourth rule of Table 2 there is a transition $u$ with $\ell(u) = \tau$ and $\pi u = \pi v | \pi w \leq M_k$ and $\pi u \cap \pi u_i = \emptyset$ for all $i > k$. So $\pi$ is $\tau$-enabled.

**Observation 2** Let $dec(P|\setminus c)|w$. Then $M$ has the form $dec(P'|c)$ and we have $dec(P)|v|dec(P')$ for $v \in T$ with $\ell(v) = \ell(u)$, $\pi u = \pi v|c$ and $\pi u = \pi v^*|c$.

For each such transition $u$, the transition $v$ discovered above is called the projection of $u$. Hence any path $\pi$ starting from a marking $dec(P)|c$ can be uniquely decomposed into a path $\pi'$ starting from $dec(P)$, notation $\pi \Rightarrow \pi'|c$. The path $\pi'$ fires all the projections of the transitions in $\pi$, in order.

**Lemma 13** If $\pi \Rightarrow \pi'|c$ and $\pi'$ is $\alpha$-enabled with $c \neq \alpha \neq \bar{c}$, then $\pi$ is $\alpha$-enabled.

**Proof** Let $\pi' = M_0|u_1|M_1|u_2|M_2|u_3|\ldots$ be $\alpha$-enabled. Then there is a $k \geq 0$ and a transition $v$ with $\ell(v) = \alpha$ such that $M_k[v]$ and $\pi v \cap \pi u_i = \emptyset$ for all $i > k$. Let $\pi = M_0|u_1|M_1|u_2|M_2|u_3|\ldots$. The marking $M_k$ has the form $dec(P_k|\setminus c) = dec(P_k)|c$ with $M_k = dec(P_k)$. Since $\pi v \subseteq dec(P_k)$ and $c \neq \alpha \neq \bar{c}$, by Table 2 there exists a transition $v'$ with $\ell(v') = \alpha$ and $\pi v' = \pi v|c \leq dec(P_k)|c = M_k$. So $M_k[v']$. Moreover, since $\pi v \cap \pi u_i = \emptyset$ for all $i > k$ we have $\pi v' \cap \pi u_i = \emptyset$ for all $i > k$. It follows that $\pi$ is $\alpha$-enabled.

□
Observation 3 Let \( \text{dec}(P[f]) \) [\( u \)] \( M \). Then \( M \) has the form \( \text{dec}(P'[f]) \) and we have \( \text{dec}(P) \) [\( v \)] \( \text{dec}(P') \) for a \( v \in T \) with \( f(\ell(v)) = \ell(u) \), \( \bullet u = \bullet v[f] \) and \( u^* = v^*[f] \).

For each such transition \( u \), the transition \( v \) discovered above is called a projection of \( u \)—it need not be unique. Hence any path \( \pi \) starting from a marking \( \text{dec}(P)c \) can be decomposed into a path \( \pi' \) starting from \( \text{dec}(P) \), notation \( \pi \in \pi'[f] \). The path \( \pi' \) fires projections of the transitions in \( \pi \), in order.

Lemma 14 If \( \pi \in \pi'[f] \) and \( \pi' \) is \( \alpha \)-enabled, then \( \pi \) is \( f(\alpha) \)-enabled.

Proof Just as the proof of Lemma [13]

Observation 4 Let \( \pi \) be a path in the unmarked Petri net of \( \text{CCS}^1 \). Then \( \pi \Rightarrow \pi_1|\pi_2 \) implies that \( \pi_1|\pi_2 \) is a decomposition of \( \pi \) (c.f. Page [12]). Likewise, if \( \pi \Rightarrow \pi' \) or \( \pi \in \pi'[f] \) then \( \pi' \) is a decomposition of \( \pi \).

Proposition 1 Let \( \pi \) be a path in the unmarked Petri net of \( \text{CCS}^1 \). If \( Y \) includes all actions \( \alpha \in \text{Act} \) for which \( \pi \) is \( \alpha \)-enabled, and \( Y \subseteq H \), then \( \pi \) is \( Y \)-just.

Proof Define a path \( \eta \) in the LTS of \( \text{CCS}^1 \) to be \( Y \)-just\( c \), for \( Y \subseteq \text{Act} \), if \( \eta \) has the form \( \pi \) for \( \pi \) a path in the unmarked Petri net of \( \text{CCS}^1 \), and \( Y \) includes all actions \( \alpha \in \text{Act} \) for which \( \pi \) is \( \alpha \)-enabled. Note that if \( \eta \) is \( Y \)-just\( c \), it is also \( Y \)-just\( c \) for any \( Y \subseteq Y' \subseteq \text{Act} \). We show that the family of predicates \( Y \)-just\( c \), for \( Y \subseteq H \), satisfies the five requirements of Definition [1]

- Let \( \pi \) be a finite \( Y \)-just\( c \) path. Suppose the last state \( Q \) of \( \tilde{\pi} \) admits an action \( \alpha \notin Y \). Then, using Theorem [3] the last marking \( \text{dec}(Q) \) of \( \pi \) enables a transition labelled \( \alpha \). Thus \( \pi \) is \( \alpha \)-enabled, contradicting the \( Y \)-just\( c \) of \( \pi \).

- Suppose \( \pi \) is a \( Y \)-just\( c \) path of a process \( P|Q \) with \( Y \subseteq H \). Then \( Y \) includes all actions \( \alpha \in \text{Act} \) for which \( \pi \) is \( \alpha \)-enabled. Let \( \pi_1 \) and \( \pi_2 \) be the paths such that \( \pi \Rightarrow \pi_1|\pi_2 \). By Observation [4] \( \tilde{\pi} \) can be decomposed into the paths \( \tilde{\pi}_1 \) of \( P \) and \( \tilde{\pi}_2 \) of \( Q \). Let \( X \subseteq \text{Act} \) be the set of actions \( \alpha \) for which \( \pi_1 \) is \( \alpha \)-enabled, and let \( Z \subseteq \text{Act} \) be the set of actions \( \alpha \) for which \( \pi_2 \) is \( \alpha \)-enabled. By definition, \( \tilde{\pi}_1 \) is \( X \)-just\( c \) and \( \tilde{\pi}_2 \) is \( Z \)-just\( c \).

If \( \pi_1 \) is \( \alpha \)-enabled then \( \pi \) is \( \alpha \)-enabled by Lemma [11]. This implies that \( X \subseteq Y \). In the same way it follows that \( Z \subseteq Y \). Now suppose \( X \cap Z \neq \emptyset \). Then \( \pi_1 \) is \( \alpha \)-enabled and \( \pi_2 \) is \( \alpha \)-enabled, for some \( \alpha \in H \). So, by Lemma [12] \( \pi \) is \( \alpha \)-enabled, in contradiction with \( \alpha \notin Y \subseteq H \). We therefore conclude that \( X \cap Z = \emptyset \).

- Suppose \( \pi \) is a \( Y \)-just\( c \) path of a process \( P \). Then \( Y \) includes all actions \( \alpha \in \text{Act} \) for which \( \pi \) is \( \alpha \)-enabled. Let \( \pi' \) be the path such that \( \pi \Rightarrow \pi' | c \). By Observation [4] \( \tilde{\pi} \) is a decomposition of the path \( \tilde{\pi}' \) of \( P \). Let \( X \subseteq \text{Act} \) be the set of actions \( \alpha \) for which \( \pi' \) is \( \alpha \)-enabled. If \( \pi' \) is \( \alpha \)-enabled with \( c \neq \alpha \neq \bar{c} \) then \( \pi \) is \( \alpha \)-enabled by Lemma [14]. This implies that \( X \setminus \{c, \bar{c}\} \subseteq Y \) and hence \( X \subseteq H \). It follows that \( \pi' \) is \( X \)-just\( c \), and hence \( Y \cup \{c, \bar{c}\} \)-just\( c \).

- Suppose \( \pi \) is a \( Y \)-just\( c \) path of a process \( P[f] \). Then \( Y \) includes all actions \( \alpha \in \text{Act} \) for which \( \pi \) is \( \alpha \)-enabled. Let \( \pi' \) be a path such that \( \pi \in \pi'[f] \). By Observation [4] \( \tilde{\pi} \) is a decomposition of the path \( \tilde{\pi}' \) of \( P \). Let \( X \subseteq \text{Act} \) be the set of actions \( \alpha \) for which \( \pi' \) is \( \alpha \)-enabled. If \( \pi' \) is \( \alpha \)-enabled then \( \pi \) is \( f(\alpha) \)-enabled by Lemma [14]. This implies that \( f(X) \subseteq Y \). It follows that \( \pi' \) is \( X \)-just\( c \), and hence \( f^{-1}(Y) \)-just\( c \).
Suppose \( \pi' \) is a suffix of an \( Y \)-just \( \pi \) path \( \pi \). Then \( Y \) includes all actions \( \alpha \in \text{Act} \) for which \( \pi \) is \( \alpha \)-enabled, and thus all \( \alpha \) for which \( \pi' \) is \( \alpha \)-enabled. Hence \( \pi' \) is \( Y \)-just \( \pi \).

Since \( Y \)-justness is the largest family of predicates that satisfies those requirements, \( Y \)-justness implies \( Y \)-justness. \( \square \)

\textbf{Corollary 1} Let \( \pi \) be a path starting from \( \text{dec}(P) \) in the unmarked Petri net of \( \text{CCS}!' \). If \( \tilde{\pi} \) is complete, then so is \( \tilde{\pi} \). Moreover, if \( \tilde{\pi} \) is \( \alpha \)-enabled, for \( \alpha \in \mathcal{A} \), then so is \( \pi \). \( \square \)

\textbf{Proof of Theorem 1} Suppose there does exist a CCS’ expression \( F \) as considered in Theorem 1. Then it suffices to show that \( \llbracket F \rrbracket \) is a net \( N \) as considered in Theorem 9. Thus, we show that \( \llbracket F \rrbracket \) satisfies the four properties of Theorem 9.

1. Let \( \pi \) be a complete path of \( \llbracket F \rrbracket \) that has finitely many occurrences of \( r_i \).
   By Lemma 10 \( \tilde{\pi} \) is a path of \( F \) that has finitely many occurrences of \( r_i \). By Corollary 1 it is complete. By Requirement 1 of Theorem 1, \( \tilde{\pi} \) is \( r_i \)-enabled. So by Corollary 1, \( \pi \) is \( r_i \)-enabled.

2. Let \( \pi \) be a complete path of \( \llbracket F \rrbracket \). Then \( \tilde{\pi} \) is a complete path of \( \llbracket F \rrbracket \). By Requirement 2 of Theorem 1 on \( \tilde{\pi} \), each \( r_i \) is followed by a \( t_i \). Using Lemma 10, the same holds for \( \pi \).

3. Let \( \pi \) be a finite path of \( \llbracket F \rrbracket \). Then \( \tilde{\pi} \) is a path of \( F \). By Requirement 3 of Theorem 1 on \( \tilde{\pi} \), and thus on \( \pi \), are no more occurrences of \( t_i \) than of \( r_i \).

4. Let \( \pi \) be a path of \( \llbracket F \rrbracket \), featuring two occurrences of \( t_i \) and \( t_j \) \((i, j \in \{1, 2\})\).
   These occurrences also occur on \( \tilde{\pi} \). By Requirement 4 of Theorem 1 an action \( e \) occurs between them. \( \square \)

\textbf{19 Concluding Remarks}

This paper presented a simple fair scheduler—one that in suitable variations occurs in many distributed systems—of which no implementation can be expressed in CCS. In particular, Dekker’s and Peterson’s mutual exclusion protocols cannot be rendered correctly in CCS. These conclusions remain true if CCS is extended with progress and certain fairness assumptions, namely justness as presented in this paper. However, as shown in [13], it is possible to correctly render Dekker’s protocol—and thereby a fair scheduler—in CCS enriched with a stronger fairness assumption. We argue, however, that such fairness assumptions should not be made lightly, as in certain contexts they allow the derivation of false results.

It does not appear hard to extend CCS with an operator that enables expressing this fair scheduler without relying on a fairness assumption. In [29] for instance we give a simple specification of a fair scheduler in an extension of CCS with broadcast communication. In [15] it is shown that it suffices (for the correct specification of Dekker’s algorithm) to extend a CCS-like process algebra with non-blocking reading actions. A priority mechanism [12] would also be sufficient.

Let \( \triangleright \) for instance be a \( + \)-like operator that schedules an action from its left argument if possible, and otherwise runs its right argument. Then \( F_1 \), with

\[
F_1 \overset{\text{def}}{=} (r_1.t_1.e.F_2) \triangleright (\tau.F_2) \quad \text{and} \quad F_2 \overset{\text{def}}{=} (r_2.t_2.e.F_1) \triangleright (\tau.F_1)
\]
appears to be a fair scheduler. Here $r_1$ is the CCS-context specified in Section 13.

$F$ is basically a round-robin scheduler which checks whether $r_1$ is enabled; if so, it performs the sequence $r_1, t_1, e$; if not, it does an internal action and tries to perform $r_2$.

An interesting question is what kind of extension of CCS is needed to enable specifying all processes of this kind. It appears that the formalism CCS+LTL that we employed in Section 11 to specify our fair scheduler can be used to specify a wide range of similar systems. Such a specification combines a CCS specification with a fairness component, consisting of a set of LTL formulas that narrows down the complete paths of the specified process. An intriguing challenge is to find an extension of CCS, say by means of extra operators, that makes the fairness component redundant, i.e. an extension such that any CCS+LTL process can be expressed in the extended CCS without employing a fairness component.

For certain properties of the form $(\bigvee_i \text{GF} a_i) \implies (\bigvee_j \text{GF} b_j)$ where the $a_i$ and $b_i$ are action occurrences—hence for specific strong fairness properties—one can define a fairness operator that transforms a given LTS into a LTS that satisfies the property [53]. This is done by eliminating all the paths that do not satisfy the property via a carefully designed parallel composition. The fairness operator can be expressed in a variant of the process algebra CSP. The question above asks whether something similar can be done, in a more expressive process algebra, for arbitrary LTL properties, or perhaps for a larger class of fairness properties.

Acknowledgements We gratefully thank the anonymous referees. Their reports showed deep insights in the material, and helped a lot to improve the quality of the paper. In particular, the link between our fair scheduler and Peterson’s mutual exclusion protocol was made by one of the referees.

References

1. Aceto, L., Ingólfsdóttir, A., Larsen, K.G., Srba, J.: Modelling mutual exclusion algorithms. In: Reactive Systems: Modelling, Specification and Verification, pp. 142–158. Cambridge University Press (2007). doi:10.1017/CBO9780511814105.008
2. Apt, K.R., Francez, N., Katz, S.: Appraising fairness in languages for distributed programming. Distributed Computing 2(4), 226–241 (1988). doi:10.1007/BF01872848
3. Baeten, J.C.M., Bergstra, J.A.: Discrete time process algebra. Formal Aspects of Computing 8(2), 188–208 (1996). doi:10.1007/BF01214556
4. Baeten, J.C.M., Bergstra, J.A., Klooster, J.W.: On the consistency of Koomen’s fair abstraction rule. Theoretical Computer Science 51(1/2), 129–176 (1987). doi:10.1016/0304-3975(87)90052-1
5. Baeten, J.C.M., Bergstra, J.A., Klooster, J.W.: Ready-trace semantics for concrete process algebra with the priority operator. Computer Journal 30(6), 498–506 (1987). doi:10.1093/comjnl/30.6.498
6. Baeten, J.C.M., Luttik, B., van Tilburg, P.: Reactive Turing machines. In: O. Owe, M. Stef sen, J.A. Telle (eds.) Fundamentals of Computation Theory, pp. 348–359 (2011). doi:10.1007/978-3-642-22953-4_30
7. Bergstra, J.A., Klooster, J.W.: Algebra of communicating processes. In: J.W. de Bakker, M. Hazewinkel, J.K. Lenstra (eds.) Mathematics and Computer Science, CWI Monograph 1, pp. 89–138. North-Holland (1986)
8. Bergstra, J.A., Klooster, J.W.: Verification of an alternating bit protocol by means of process algebra. In: W. Bibel, K.P. Jantke (eds.) Mathematical Methods of Specification and Synthesis of Software Systems ’85, LNCS, vol. 215, pp. 9–23. Springer (1986). doi:10.1007/3-540-16444-8_1
9. Bolognesi, T., Brinksmä, E.: Introduction to the ISO specification language LOTOS. Computer Networks 14, 25–59 (1987). doi:10.1016/0169-7552(87)90085-7
10. Bouali, A.: Weak and branching bisimulation in Fctool. Research Report RR-1575, Inria-Sophia Antipolis (1992). URL https://hal.inria.fr/inria-00074985/document

11. Brookes, S.D., Hoare, C.A.R., Roscoe, A.W.: A theory of communicating sequential processes. J. ACM 31(5), 560–599 (1984). doi:10.1145/582.833

12. Cleaveland, R., Lüttgen, G., Natarajan, V.: Priority in process algebra. In: J.A. Bergstra, A. Ponse, S.A. Smolka (eds.) Handbook of Process Algebra, chap. 12, pp. 711–765. Elsevier (2001). doi:10.1016/B978-044452830-9/50030-8

13. Corradini, F., Di Berardini, M.R., Vogler, W.: Fairness of actions in system computations. Acta Informatica 43(2), 73–130 (2006). doi:10.1007/s00236-006-0011-2

14. Corradini, F., Di Berardini, M.R., Vogler, W.: Liveness of a mutex algorithm in a fair process algebra. Acta Informatica 46(3), 209–235 (2009). doi:10.1007/s00236-009-0092-9

15. Corradini, F., Di Berardini, M.R., Vogler, W.: Time and fairness in a process algebra with non-blocking reading. In: M. Nielsen, A. Kucera, P.B. Miltersen, C. Palamidessi, P. Tuma, F.D. Valencia (eds.) Theory and Practice of Computer Science (SOFSEM’09), LNCS, vol. 5404, pp. 193–204. Springer (2009). doi:10.1007/978-3-540-95891-8_20

16. Corradini, F., Vogler, W., Jenner, L.: Comparing the worst-case efficiency of asynchronous systems with PAFA. Acta Informatica 38(11/12), 735–792 (2002). doi:10.1007/s00236-002-0094-3

17. Costa, G., Stirling, C.: A fair calculus of communicating systems. Acta Informatica 21, 417–441 (1984). doi:10.1007/BF00271640

18. Costa, G., Stirling, C.: Weak and strong fairness in CCS. Information and Computation 73(4), 207–244 (1987). doi:10.1016/0890-5401(87)90013-7

19. Degano, P., De Nicola, R., Montanari, U.: CCS is an (augmented) contact free C/E system. In: M.V. Zilli (ed.) Mathematical Models for the Semantics of Parallelism, LNCS, vol. 280, pp. 144–165. Springer (1987). doi:10.1007/3-540-19419-8_13

20. Dijkstra, E.W.: Over de sequentialiteit van procesbeschrijvingen (1962 or 1963). URL http://www.cs.utexas.edu/users/EWD/ewd00xx/EWD35.PDF. Circulated privately

21. Dijkstra, E.W.: Cooperating sequential processes. In: F. Genuys (ed.) Programming Languages: NATO Advanced Study Institute, pp. 43–112. Academic Press (1968)

22. Esparza, J., Bruns, G.: Trapping mutual exclusion in the box calculus. Theoretical Computer Science 153(1-2), 95–128 (1996). doi:10.1016/0304-3975(95)00119-0

23. Fehnker, A., van Glabbeek, R.J., Höfner, P., McIver, A.K., Portmann, M., Tan, W.L.: A process algebra for wireless mesh networks used for modelling, verifying and analysing AODV. Tech. Rep. 5513, NICTA (2013). Available at http://arxiv.org/abs/1312.7645

24. Francez, N.: Fairness. Springer (1986). doi:10.1007/978-1-4612-4886-6

25. Gabbay, D.M., Pnueli, A., Shelah, S., Stavi, J.: On the temporal analysis of fairness. In: P.W. Abrahams, R.J. Lipton, S.R. Bourne (eds.) Principles of Programming Languages (POPL ’80), pp. 163–173. ACM (1980). doi:10.1145/657446.657462

26. van Glabbeek, R.J.: On specifying timeouts. In: L. Aceto, A.D. Gordon (eds.) Short Contributions from the Workshop on Algebraic Process Calculi: The First Twenty Five Years and Beyond, ENTCS, vol. 162, pp. 112–113. Elsevier (2005). doi:10.4204/EPTCS.89.7

27. van Glabbeek, R.J., Goltz, U., Schicke, J.W.: Abstract processes of place/transition systems. Information Processing Letters 111(13), 626–633 (2011). doi:10.1016/j.ipl.2011.03.013

28. van Glabbeek, R.J., Höfner, P.: Progress, fairness and justness in process algebra. CoRR abs/1501.03268 (2015). URL http://arxiv.org/abs/1501.03268

29. van Glabbeek, R.J., Vaandrager, F.W.: Petri net models for algebraic theories of concurrency. In: J.W. Bakker, A.J. Nijman, P.C. Treleaven (eds.) Parallel Architectures and Languages Europe (PARLE’97), Vol. II: Parallel Languages, LNCS, vol. 259, pp. 224–242. Springer (1987). doi:10.1007/3-540-17965-3_13

30. Goldin, D.Q., Smolka, S.A., Attie, P.C., Sondergerger, E.L.: Turing machines, transition systems, and interaction. Information and Computation 194(2), 101–128 (2004). doi:10.1016/j.ic.2004.07.002

31. Gorla, D.: Towards a unified approach to encodability and separation results for process calculi. Information and Computation 208(9), 1031–1053 (2010). doi:10.1016/j.ic.2010.05.002
59. Valmari, A., Setälä, M.: Visual verification of safety and liveness. In: M. Gaudel, J. Woodcock (eds.) Industrial Benefit and Advances in Formal Methods (FME’96), LNCS, vol. 1051, pp. 228–247. Springer (1996). doi:10.1007/3-540-60973-3_90
60. Vogler, W.: Efficiency of asynchronous systems, read arcs, and the MUTEX-problem. Theor. Comput. Sci. 275(1-2), 589–631 (2002). doi:10.1016/S0304-3975(01)00300-0
61. Walker, D.J.: Automated analysis of mutual exclusion algorithms using CCS. Formal Aspects of Computing 1(1), 273–292 (1989). doi:10.1007/BF01887209
62. Wegner, P.: Why interaction is more powerful than algorithms. Communications of the ACM 40(5), 80–91 (1997). doi:10.1145/253769.253801