THE UNIVERSITY of EDINBURGH

Edinburgh Research Explorer

Modelling Mutual Exclusion in a Process Algebra with Time-outs

Citation for published version:
van Glabbeek, R 2023, 'Modelling Mutual Exclusion in a Process Algebra with Time-outs', *Information and Computation*, vol. 294, 105079, pp. 1-36. https://doi.org/10.1016/j.ic.2023.105079

Digital Object Identifier (DOI):
10.1016/j.ic.2023.105079

Link:
Link to publication record in Edinburgh Research Explorer

Document Version:
Publisher's PDF, also known as Version of record

Published In:
Information and Computation

General rights
Copyright for the publications made accessible via the Edinburgh Research Explorer is retained by the author(s) and / or other copyright owners and it is a condition of accessing these publications that users recognise and abide by the legal requirements associated with these rights.

Take down policy
The University of Edinburgh has made every reasonable effort to ensure that Edinburgh Research Explorer content complies with UK legislation. If you believe that the public display of this file breaches copyright please contact openaccess@ed.ac.uk providing details, and we will remove access to the work immediately and investigate your claim.

Download date: 26. Apr. 2025

Information and Computation 294 (2023) 105079

Contents lists available at ScienceDirect

Information and Computation

journal homepage: www.elsevier.com/locate/yinco

Modelling mutual exclusion in a process algebra with time-outs

Rob van Glabbeek [a,b,c,*,1]

[a] Data61, CSIRO, Sydney, Australia
[b] School of Informatics, University of Edinburgh, UK
[c] School of Computer Science and Engineering, University of New South Wales, Sydney, Australia

ARTICLE INFO

Article history:
Received 25 June 2021
Received in revised form 15 June 2023
Accepted 3 August 2023
Available online 10 August 2023

Keywords:
Mutual exclusion
Safe registers
Overlapping reads and writes
Atomicity
Speed independence
Reactive temporal logic
Kripke structures
Progress
Justness
Fairness
Safety properties
Blocking
Fair schedulers
Process algebra
CCS
Time-outs
Labelled transition systems
Petri nets
Asymmetric concurrency relations
Peterson's protocol

ABSTRACT

I show that in a standard process algebra extended with time-outs one can correctly model mutual exclusion in such a way that starvation-freedom holds without assuming fairness or justness, even when one makes the problem more challenging by assuming memory accesses to be atomic. This can be achieved only when dropping the requirement of speed independence.

© 2023 The Author(s). Published by Elsevier Inc. This is an open access article under the CC BY license (http://creativecommons.org/licenses/by/4.0/).

Contents

* Correspondence to: School of Informatics, University of Edinburgh, UK.
E-mail address: rvg@cs.stanford.edu.

[1] Supported by Royal Society Wolfson Fellowship RSWF\R1\221008.

https://doi.org/10.1016/j.ic.2023.105079
0890-5401/© 2023 The Author(s). Published by Elsevier Inc. This is an open access article under the CC BY license (http://creativecommons.org/licenses/by/4.0/).

1. Introduction

A *mutual exclusion protocol* mediates between competing processes to make sure that at any time at most one of them visits a so-called *critical section* in its code. Such a protocol is *starvation-free* when each process that intends to enter its critical section will eventually be allowed to do so.

As shown in [39,60,32], it is fundamentally impossible to correctly model a mutual exclusion protocol as a Petri net or in standard process algebras, such as CCS [47], CSP [8,37] or ACP [3,19], unless starvation-freedom hinges on a fairness assumption. The latter, in the view of [32], does not provide an adequate solution, as fairness assumptions are in many situations unwarranted and lead to false conclusions.

In [17] a correct process-algebraic rendering of mutual exclusion is given, but only after making two important modifications to standard process algebra. The first involves making a justness assumption. Here *justness* [31,33] is an alternative to fairness, in some sense a much weaker form of fairness—meaning weaker than weak fairness.[2] Unlike (strong or weak)

[2] Justness is the assumption that when a certain activity *can* occur in a distributed system, eventually either it *will* occur, or one of the resources needed to perform this activity is used for some other purpose. This is illustrated by the forthcoming Examples 3.3–3.5; justness is a strong enough assumption to conclude that Bart gets a beer in Example 3.5, but to ensure this even for Example 3.3 one needs the stronger assumption of weak fairness.

fairness, its use typically *is* warranted and does not lead to false conclusions. The second modification is the addition of a nonstandard construct—*signals*—to CCS, or any other standard process algebra. Interestingly, both modifications are necessary; merely assuming justness, or merely adding signals, is insufficient.

A similar process-algebraic rendering of mutual exclusion was given earlier in [12], using a fairness assumption proposed in [11] under the name *fairness of actions*. In [33] fairness of actions (there called *fairness of events*) was seen to coincide with justness.

Bouwman [6,7] points out that it is possible to correctly model mutual exclusion without adding signals to the language at all, instead reformulating the justness requirement in such a way that it effectively turns some actions into signals. Since the justness assumption was fairly new, and needed to be carefully defined to describe its interaction with signals anyway, redefining it to better capture read actions in mutual exclusion protocols is a plausible solution.

Yet justness is essential in all the above approaches. This may be seen as problematic, because large parts of the foundations of process algebra are incompatible with justness, and hence need to be thoroughly reformulated in a justness-friendly way. This is pointed out in [26].[3]

In [32], the inability to correctly capture mutual exclusion in CCS and related process algebras was seen as a sign that these process algebras lack some degree of universal expressiveness, rather than as a statement about the impossibility of mutual exclusion. The repairs in [12,17,6,7] seek to rectify this lack of expressiveness, either by considering language extensions, or by changing the definition of justness for the language. My presentation [24] took a different perspective, and claimed that the impossibility results of [39,60,32] can be seen as saying something about the real world, rather than about formalisms we use to model it. The argument rests on two crucial features of mutual exclusion protocols that I call *atomicity* and *speed independence*. Instead of protocol features they can also be seen as assumptions on the hardware on which the mutual exclusion protocol will be running.

Atomicity is the assumption that *memory accesses such as reads and writes take a positive amount of time, yet two such accesses to the same store or register cannot overlap in time, so that a second memory access can take place only after a first access is completed*.[4] Speed independence says that nothing may be assumed about the relative speed of processes competing for access to the critical section, or for read/write access to some register. In particular, if two processes are engaged in a race, and one of them has nothing else to do but performing the action that wins the race, whilst the other has a long list of tasks that must be done first, it may still happen that the other process wins.

When rejecting solutions to the mutual exclusion problem that are merely probabilistically correct, or where starvation-freedom hinges on a fairness assumption, [24] claims, although without written evidence, that when assuming atomicity as well as speed independence, mutual exclusion is impossible. Section 22 of the present paper illustrates and substantiates this claim for Peterson's mutual exclusion protocol.

In [33] the notion of justness from [31] was reformulated in terms of a concurrency relation $\frown\!\!\bullet$ between the transitions in a labelled transition system. This relation may be inherited from a similar relation between the transitions of a Petri net or the instructions in the pseudocode of protocol descriptions. Here $t \not\frown\!\!\bullet\, u$ means that transition or instruction u uses (takes away) a resource that is needed to perform transition or instruction t, so that if u occurs prior to, or instead of, t, it is not possible for t to commence before u is finished.[5] The definitions of justness from [31] and [33] were shown equivalent in [25].

The assumption of atomicity can be formulated directly in terms of the concurrency relation $\frown\!\!\bullet$. It says about read or write instructions or transitions ℓ and m,

$$\text{if } \ell \text{ and } m \text{ access the same register then } \ell \not\frown\!\!\bullet\, m \text{ and } m \not\frown\!\!\bullet\, \ell.$$

In other words, in case ℓ and m try to access the same register in parallel, and m wins the race for access to this register, ℓ cannot take place until m is completed. The case that is relevant for the mutual exclusion problem is where ℓ writes and m reads.

I see only two alternatives to $\ell \not\frown\!\!\bullet\, m \wedge m \not\frown\!\!\bullet\, \ell$. One is that the memory accesses ℓ and m overlap in time. This possibility has been investigated by Lamport [42], who assumes that a read action that overlaps with a write on the same register can return any possible value of that register. Since the return of an unexpected value increases the set of possible behaviours of a mutual exclusion protocol, Lamport implicitly takes the position that assuming overlap of actions makes the mutual exclusion problem more challenging than assuming atomicity. Yet, he shows that his bakery algorithm [42] constitutes an

[3] This problem has however been mitigated in [7], where a standard process algebra is used in a way that is compatible with justness. This led to the successful verification of Peterson's mutual exclusion protocol under the assumption of justness, using the mCRL2 toolset [9]. The price to be paid for this is that more information needs to be encoded in the actions that are used as transition labels, and that many transitions that in classical models would be labelled with the hidden action τ, need now have a visible label. The latter inhibits state-space reduction techniques that abstract from such actions.

[4] This appears to be a consequence of seeing reads and writes as *atomic actions*. In this paper "atomicity" refers to the above assumption; it will not include the case where one memory access may abort or interrupt another, even when this entails that memory accesses cannot overlap in time. Subtly different notions of atomicity are that of an *atomic register*, which merely behaves *as if* its reads and writes are atomic, and that of an *atomic transaction* in database systems, which needs to either complete, or be rolled back completely.

[5] In standard Petri nets, standard process algebras, and many other models of concurrency, the concurrency relation is symmetric, in the sense that $t \not\frown\!\!\bullet\, u \Rightarrow u \not\frown\!\!\bullet\, t$. Exceptions to symmetry are rare, but they will play a vital rôle in parts of this paper. They can occur when a transition needs a resource (like sunshine) without blocking it for others, or when a transition uses a resource when available, without actually needing it.

entirely correct solution. It moreover trivially is speed independent. However, according to [24] atomicity is the more challenging assumption, as when assuming overlap a correct speed-independent solution exists, and when assuming atomicity it does not.

The second alternative to $\ell \not\frown m \wedge m \not\frown \ell$ retains the assumption that memory accesses to the same register cannot overlap in time, but assumes write actions to have priority over reads. A write simply aborts a read that happens to be in progress, which can restart after the write is over. Following [12], I refer to this assumption as *non-blocking reading*. When ℓ is a write action and m a read of the same register, it stipulates that $\ell \smile m$, yet $m \not\frown \ell$. This yields an asymmetric concurrency relation, which was not foreseen in classical treatments of concurrency [50,35,5,62,34,14,51].

The assumption of speed independence is built in in CCS and Petri nets, in the sense that any correct mutual exclusion protocol formalised therein is automatically speed independent. This is because these models lack the expressiveness to make anything dependent on speed. In Section 4.4, following [26], I define a (symmetric) concurrency relation between Petri net transitions and between CCS transitions that is consistent with the work in [50,35,5,62,34,14,51]. It always yields $\ell \not\frown m$ when ℓ and m both access the same register. When taking this concurrency relation as an integral part of semantics of CCS or Petri nets, it follows that also the assumption of atomicity is built in in these frameworks. This makes the impossibility results of [39,60,32] special cases of the impossibility claim from [24]. The latter can be seen as a generalisation of the former that is not dependent on a particular modelling framework.

The process algebras of [12] and [17] model the possibility of non-blocking reading. This enables a correct rendering of speed independent mutual exclusion without resorting to a fairness assumption. The first such correct model of exclusion occurs in Vogler [60] in terms of Petri nets extended with read arcs; the latter enable the modelling of non-blocking reading. The correct modelling of mutual exclusion within CCS as proposed by Bouwman [6] also exploits non-blocking reading. The justness assumption as formulated by Bouwman can in retrospect be seen as an instance of justness as defined in [33], but based on a concurrency relation $\smile$ between CCS transitions that essentially differs from the one in Section 4.4, and that is not consistent with the work in [50,35,5,62,34,14,51], although it is entirely consistent with the interleaving semantics of CCS given by Milner [47]. The claim in [32,17] that mutual exclusion cannot be rendered satisfactory in CCS holds only when seeing the concurrency relation of Section 4.4 (or the resulting notion of justness) as an integral part of this language, and hence is not in contradiction with the findings of Bouwman [6].

In [29] I extended standard process algebra with a time-out operator, thereby increasing its absolute expressiveness, while remaining within the realm of untimed process algebra, in the sense that the progress of time is not quantified. The present paper shows that the addition of time-outs to standard process algebra makes it possible to correctly model mutual exclusion under the assumption of atomicity, such that starvation-freedom holds without assuming fairness. My witness for this claim will be a model of Peterson's mutual exclusion protocol. In view of the above, this model will not be speed independent.

Moreover, starvation-freedom can be shown to hold, not only without assuming fairness, but even without assuming justness. Instead, one should make the assumption called *progress* in [33], which is weaker than justness, uncontroversial, unproblematic, and made (explicitly or implicitly) in virtually all papers dealing with issues like mutual exclusion. In contrast, [24] claims that even when dropping atomicity it is not possible to correctly model mutual exclusion in a speed-independent way without at least assuming justness to obtain starvation-freedom. Section 16/17 of the present paper illustrates and substantiates also that claim for Peterson's mutual exclusion protocol.

Reading guide Part IV of this paper shows how Peterson's mutual exclusion protocol can be modelled in an extension of CCS with time-outs. This process algebra assumes atomicity, as one has $\ell \not\frown m$ whenever m and ℓ are read and write transitions on the same register. The model satisfies all basic requirements for mutual exclusion protocols, and in addition achieves starvation-freedom without assuming more than progress.

Part III recalls all impossibility claims discussed above, and illustrates or substantiates them for Peterson's mutual exclusion protocol. To make the impossibility claims precise, I have to define unambiguously what does and what does not constitute a correct mutual exclusion protocol. This happens in Part II. That part also covers *fair schedulers* [32], which are akin to mutual exclusion protocols, and were used in [32] as a stepping stone to prove the impossibility result for mutual exclusion in CCS. I formalise four requirements on fair schedulers and six on mutual exclusion protocols that in combination determine their correctness. Some of these requirements, including starvation-freedom for mutual exclusion protocols, are parametrised with an assumption such as progress, justness or fairness, that needs to be made to fulfil this requirement. This leads to a hierarchy of quality criteria for fair schedulers and mutual exclusion protocols, where the quality of such a protocol is higher when it depends on a weaker assumption. I also propose two related mutual exclusion protocols, the *gatekeeper* and the *encapsulated gatekeeper*, that meet all correctness criteria when allowing (weak) fairness as parameter in some of the requirements. As I expect most researchers in the area of mutual exclusion to agree with me that the (encapsulated) gatekeeper is not an acceptable protocol, this underpins the verdict of [32] that assuming (weak) fairness does not yield an acceptable solution.

The requirements on fair schedulers and mutual exclusion protocols in Part II are formulated in the language of *linear-time temporal logic* [53,38]. However, standard treatments of temporal logic turned out to be inadequate to formalise these requirements. For this reason, Part I presents a form of temporal logic that is adapted for the study of reactive systems, interacting with their environments through synchronisation of actions. (Reactive) temporal logic primarily applies to distributed systems formalised as states in a Kripke structure. However, it smoothly lifts to distributed systems formalised, for

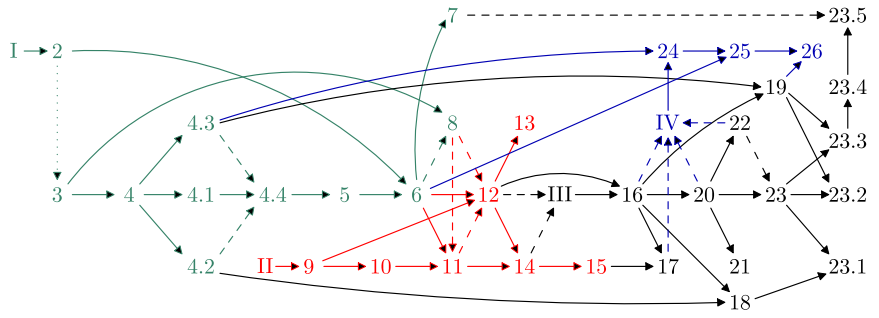


Fig. 1. Dependence relations between the sections of this paper.

instance, as states in labelled transition systems, as Petri nets, or as expressions in a process algebra like CCS. As explained in Section 4, this is achieved through canonical translations from (states in) labelled transition systems to (states in) Kripke structures, and from Petri nets or process algebra expressions to states in labelled transition systems. Assumptions such as progress, justness and fairness are gathered under the heading *completeness criteria*, as in essence they say which execution paths are regarded as complete runs of a represented system. These criteria are incorporated in reactive temporal logic. To capture justness, Section 4.4 defines a concurrency relation $\between$ on the labelled transition systems that occur in the translation steps from CCS or Petri nets to Kripke structures.

As a reading guide, I offer a table of contents and the diagram of Fig. 1. Here a dashed arrow indicates that although a concept introduced in the source section returns in the target, in spite of this the target section can be read independently of the source. The different colours mark the four parts of which this paper consists. Sections 2–6 and 9–12 are taken from [28]; the only added novelty is the treatment of the next-state operator **X** in reactive linear-time temporal logic, and the corresponding mild simplification of the requirements on fair schedulers and mutual exclusion protocols in Sections 11 and 12. Section 7, on translating reactive temporal logic into standard temporal logic, is also new here, as well as Section 8, characterising a fragment of linear-time temporal logic that denotes safety formulas. On that fragment there is no difference between reactive and standard temporal logic.

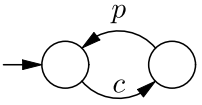
Part I. Reactive temporal logic

Whereas standard treatments of temporal logic are adequate for *closed systems*, having no run-time interactions with their environment, they fall short for *reactive systems*, interacting with their environments through synchronisation of actions. Here I present *reactive temporal logic* [28], a form of temporal logic adapted for the study of reactive systems.

2. Motivation

Labelled transition systems are a common model of distributed systems. They consist of sets of states, also called *processes*, and transitions—each transition going from a source state to a target state. A given distributed system $\mathscr{D}$ corresponds to a state P in a transition system $\mathbb{T}$—the initial state of $\mathscr{D}$. The other states of $\mathscr{D}$ are the processes in $\mathbb{T}$ that are reachable from P by following the transitions. The transitions are labelled by *actions*, either visible ones or the invisible action τ. Whereas a τ-labelled transition represents a state-change that can be made spontaneously by the represented system, a-labelled transitions, for $a \neq \tau$, merely represent potential activities of $\mathscr{D}$, for they require cooperation from the *environment* in which $\mathscr{D}$ will be running, sometimes identified with the *user* of system $\mathscr{D}$. A typical example is the acceptance of a coin by a vending machine. For this transition to occur, the vending machine should be in a state where it is enabled, i.e., the opening for inserting coins should not be closed off, but also the user of the system should partake by inserting the coin.

Consider a vending machine that alternatingly accepts a coin (c) and dispenses a pretzel (p). Its labelled transition system is depicted on the right. In standard temporal logic one can express that each action c is followed by p: whenever a coin is inserted, a pretzel will be dispensed. Aligned with intuition, this formula is valid for the depicted system. However, by symmetry one obtains the validity of a formula saying that each p is followed by a c: whenever a pretzel is dispensed, eventually a new coin will be inserted. But that clashes with intuition.



In [28] I enriched temporal logic judgements $P \models \varphi$, saying that system P satisfies formula φ, with a third argument B, telling which actions can be blocked by the environment (by failing to act as a synchronisation partner) and which cannot. When stipulating that the coin needs cooperation from a user, but producing the pretzel does not, the two temporal judgements can be distinguished, and only one of them holds. I also introduced a fourth argument CC—a completeness

criterion—that incorporates progress, justness and fairness assumptions employed when making a temporal judgement. This yields statements of the form $P \models_B^{CC} \varphi$.[6]

The work in [28] builds on an earlier approach from [31], where judgements $P \models_B^{CC} \varphi$ were effectively introduced. However, there they were written $P \models \varphi$, based on the assumption that for a given application a completeness criterion CC and a set of blocking actions B would be fixed. The idea was that at the beginning of a paper employing temporal logic, a given CC and B would be declared, after which all forthcoming judgements $P \models \varphi$ would be interpreted as $P \models_B^{CC} \varphi$. The novelty of the approach in [28] is to make CC and B as variable as P and φ, so that in the description of a single system, temporal judgements with different values of CC and B can be combined.

Suppose that P is the initial state of the example above, and $\mathbf{G}(a \Rightarrow \mathbf{F}b)$ is a formula that says that each action a is followed by a b. Abstracting from the completeness criterion for the moment, one has

$$P \models_{\{c\}} \mathbf{G}(c \Rightarrow \mathbf{F}p) \qquad P \not\models_{\{c\}} \mathbf{G}(p \Rightarrow \mathbf{F}c) \qquad P \models_{\emptyset} \mathbf{G}(p \Rightarrow \mathbf{F}c)\,.$$

The first judgement says that whenever a coin is inserted, a pretzel will be dispensed, even if we operate in an environment that may never insert a coin. By taking $B = \{c\} \not\ni p$, the judgement also assumes that the environment will never block the production of a pretzel.

The second judgement says that in the same environment there is no guarantee that each production of a pretzel is followed by the insertion of another coin.

The third judgement says that if we happen to run our vending machine in an environment where the user is perpetually eager to insert a new coin, after each pretzel, acceptance of the next coin is guaranteed. This is an important correctness property of the vending machine. Without such a property the machine is rather unsatisfactory. Hence a specification of the kind of vending machine one would like to have could be a combination of the first and third judgement above. This kind of specification was not foreseen in [31].

3. Kripke structures and linear-time temporal logic

Definition 3.1. Let AP be a set of *atomic predicates*. A *Kripke structure* over AP is tuple $(S, \rightarrow, \models)$ with S a set (of *states*), $\rightarrow \subseteq S \times S$, the *transition relation*, and $\models \subseteq S \times AP$. $s \models p$ says that predicate $p \in AP$ *holds* in state $s \in S$.

Here I generalise the standard definition (see for instance [38]) by dropping the condition of *totality*, requiring that for each state $s \in S$ there is a transition $(s, s') \in \rightarrow$. A *path* in a Kripke structure is a nonempty finite or infinite sequence $s_0, s_1, \ldots$ of states, such that $(s_i, s_{i+1}) \in \rightarrow$ for each adjacent pair of states s_i, s_{i+1} in that sequence. Write $\rho \leq \pi$ if path ρ is a prefix of path π. If $\rho \leq \pi$ and ρ is finite, then $\pi \restriction \rho$ denotes the suffix of π that remains after removing the prefix ρ, but not the last state of ρ. The *length* of a path π, denoted $|\pi| \in \mathbb{N} \cup \{\infty\}$, is the number of transitions in π; for instance $l(s_0 s_1 s_2 s_3) = 3$.

A distributed system $\mathscr{D}$ can be modelled as a state s in a Kripke structure K. A run of $\mathscr{D}$ then corresponds with a path in K starting in s. Whereas each finite path in K starting from s models a *partial run* of $\mathscr{D}$, i.e., an initial segment of a (complete) run, typically not each path models a run. Therefore a Kripke structure constitutes a good model of distributed systems only in combination with a *completeness criterion* [25]: a selection of a set of paths as *complete paths*, modelling runs of the represented system.

The default completeness criterion, implicitly used in almost all work on temporal logic, classifies a path as complete iff it is infinite. In other words, only the infinite paths, and all of them, model (complete) runs of the represented system. This applies when adopting the condition of totality, so that each finite path is a prefix of an infinite path. Naturally, in this setting there is no reason to use the word "complete", as "infinite" will do. As I plan to discuss alternative completeness criteria in Section 5, I here already refer to paths satisfying a completeness criterion as "complete" rather than "infinite". Moreover, when dropping totality, the default completeness criterion is adapted to declare a path complete iff it either is infinite or ends in a state without outgoing transitions [13].

Linear-time temporal logic (LTL) [53,38] is a formalism explicitly designed to formulate properties such as the safety and liveness requirements of mutual exclusion protocols. Its syntax is

$$\varphi, \psi ::= p \mid \neg\varphi \mid \varphi \wedge \psi \mid \mathbf{X}\varphi \mid \mathbf{F}\varphi \mid \mathbf{G}\varphi \mid \psi \mathbf{U} \varphi$$

with $p \in AP$ an atomic predicate. The propositional connectives $\Rightarrow$ and $\vee$ can be added as syntactic sugar. It is interpreted on the paths in a Kripke structure. The relation $\models$ between paths and LTL formulae, with $\pi \models \varphi$ saying that the path π *satisfies* the formula φ, or that φ is *valid* on π, is inductively defined by

- $\pi \models p$, with $p \in AP$, iff $s \models p$, where s is the first state of π,
- $\pi \models \neg\varphi$ iff $\pi \not\models \varphi$,

[6] The technical development introduces ternary judgements $P \models^{CC} \varphi$ as a primitive, and obtains the quaternary judgements $P \models_B^{CC} \varphi$ by employing a completeness criterion $CC(B)$ that itself is parametrised by a set B of blocking actions.

- $\pi \models \varphi \wedge \psi$ iff $\pi \models \varphi$ and $\pi \models \psi$,
- $\pi \models \mathbf{X}\varphi$ iff $|\pi| > 0$ and $\pi_{+1} \models \varphi$, where π_{+1} is obtained from π by omitting its first state,
- $\pi \models \mathbf{F}\varphi$ iff $\pi \restriction \rho \models \varphi$ for some finite prefix ρ of π,
- $\pi \models \mathbf{G}\varphi$ iff $\pi \restriction \rho \models \varphi$ for each finite prefix ρ of π, and
- $\pi \models \psi\mathbf{U}\varphi$ iff $\pi \restriction \rho \models \varphi$ for some finite prefix ρ of π, and $\pi \restriction \zeta \models \psi$ for each $\zeta < \rho$.

In the standard treatment of LTL [53,38], judgements $\pi \models \varphi$ are pronounced only for infinite paths π. Here I apply the same definitions verbatim to finite paths as well. Extra care is needed only in the definition of the *next-state* operator $\mathbf{X}\varphi$; here the condition $|\pi| > 0$ is redundant when π is infinite. One can define a *weak next-state* operator $\mathbf{Y}\varphi$ by

- $\pi \models \mathbf{Y}\varphi$ iff $|\pi| = 0$ or $\pi_{+1} \models \varphi$, where π_{+1} is obtained from π by omitting the first state.

Now $\mathbf{Y}$ is the *dual* of $\mathbf{X}$, in the sense $\mathbf{Y}\varphi \equiv \neg\mathbf{X}\neg\varphi$ and $\mathbf{X}\varphi \equiv \neg\mathbf{Y}\neg\varphi$, just like $\mathbf{F}$ is the dual of $\mathbf{G}$. Here $\varphi \equiv \psi$ means that $(\pi \models \varphi) \Leftrightarrow (\pi \models \psi)$ for all paths π in all Kripke structures. The distinction between strong and weak next-state operators stems from [46], where $\mathbf{F}$, $\mathbf{G}$, $\mathbf{X}$ and $\mathbf{Y}$ are written $\Diamond$, $\Box$, $\bigcirc$ and $\odot$. When only infinite paths are considered, there is no difference between $\mathbf{X}$ and $\mathbf{Y}$.

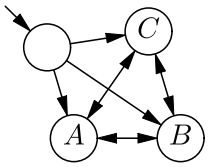
Having given meaning to judgements $\pi \models \varphi$, as a derived concept one defines when an LTL formula φ holds for a state s in a Kripke structure, modelling a distributed system $\mathscr{D}$, notation $s \models \varphi$ or $\mathscr{D} \models \varphi$. This is the case iff φ holds for all runs of $\mathscr{D}$.

Definition 3.2. $s \models \varphi$ iff $\pi \models \varphi$ for all complete paths π starting in state s.

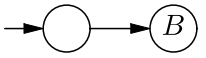
This definition depends on the underlying completeness criterion, telling which paths model actual system runs. In situations where I consider different completeness criteria, I make this explicit by writing $s \models^{CC} \varphi$, with CC the name of the completeness criterion used. When leaving out the superscript CC I refer to the default completeness criterion, defined above.

Example 3.3. Alice, Bart and Cameron stand behind a bar, continuously ordering and drinking beer. Assume they do not know each other and order individually. As there is only one bartender, they are served sequentially. Also assume that none of them is served twice in a row, but as it takes no longer to drink a beer than to pour it, each of them is ready for the next beer as soon as another person is served.

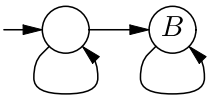
A Kripke structure of this distributed system $\mathscr{D}$ is drawn on the right. The initial state of $\mathscr{D}$ is indicated by a short arrow. The other three states are labelled with the atomic predicates A, B and C, indicating that Alice, Bart or Cameron, respectively, has just acquired a beer. When assuming the default completeness criterion, valid LTL formulae are $\mathbf{F}(A \vee C)$, saying that eventually either Alice or Cameron will get a beer, or $\mathbf{G}(A \Rightarrow \mathbf{F}\neg A)$, saying that each time Alice got a beer is followed eventually by someone else getting one. However, it is not guaranteed that Bart will ever get a beer: $\mathscr{D} \not\models \mathbf{F}B$. A counterexample for this formula is the infinite run in which Alice and Cameron get a beer alternatingly.



Example 3.4. Bart is the only customer in a bar in London, with a single bartender. He only wants one beer. A Kripke structure of this system $\mathscr{E}$ is drawn on the right. When assuming the default completeness criterion, this time Bart gets his beer: $\mathscr{E} \models \mathbf{F}B$.



Example 3.5. Bart is the only customer in a bar in London, with a single bartender. He only wants one beer. At the same time, Alice and Cameron are in a bar in Tokyo. They drink a lot of beer. Bart is not in contact with Alice and Cameron, nor is there any connection between the two bars. Yet, one may choose to model the drinking in these two bars as a single distributed system. A Kripke structure of this system $\mathscr{F}$ is drawn on the right, collapsing the orders of Alice and Cameron, which can occur before or after Bart gets a beer, into self-loops. When assuming the default completeness criterion, Bart cannot count on a beer: $\mathscr{F} \not\models \mathbf{F}B$.



4. Labelled transition systems, process algebra and Petri nets

The most common formalisms in which to present reactive distributed systems are pseudocode, process algebra and Petri nets. The semantics of these formalisms is often given through translations into labelled transition systems (LTSs), and these in turn can be translated into Kripke structures, on which temporal formulae from languages such as LTL are interpreted. These translations make the validity relation $\models$ for temporal formulae applicable to all these formalisms. A state in an LTS, for example, is defined to satisfy an LTL formula φ iff its translation into a state in a Kripke structure satisfies this formula.

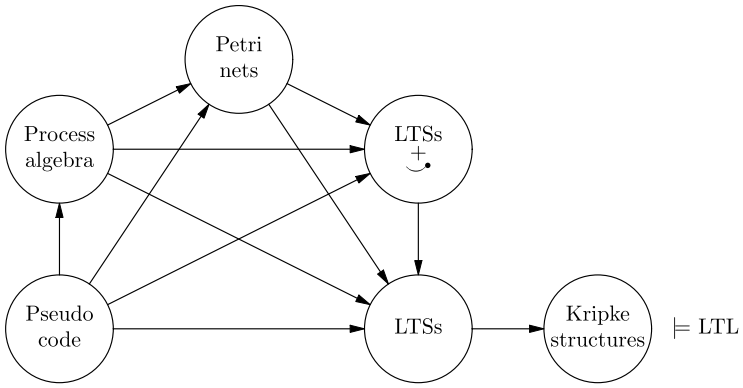


Fig. 2. Formalisms for modelling mutual exclusion protocols.

Fig. 2 shows a commutative diagram of semantic translations found in the literature, from pseudocode, process algebra and Petri nets via LTSs to Kripke structures. Each step in the translation abstracts from certain features of the formalism at its source. Some useful requirements on distributed systems can be adequately formalised in process algebra or Petri nets, and informally described for pseudocode, whereas LTSs and Kripke structures have already abstracted from the relevant information. An example will be FS1 on page 18. I also consider LTSs upgraded with a concurrency relation $\frown$ between transitions; these will be expressive enough to formalise some of these requirements.

4.1. Labelled transition systems

Definition 4.1. Let A be a set of *observable actions*, and let $Act := A \dot\cup \{\tau\}$, with $\tau \notin A$ the *hidden action*. A *labelled transition system* (LTS) over Act is a tuple $(\mathbb{P}, Tr, src, trg, \ell)$ with $\mathbb{P}$ a set (of *states* or *processes*), Tr a set (of *transitions*), $src, trg : Tr \to \mathbb{P}$ and $\ell : Tr \to Act$.

Write $s \xrightarrow{\alpha} s'$ if there is a $t \in Tr$ with $src(t) = s \in \mathbb{P}$, $\ell(t) = \alpha \in Act$ and $trg(t) = s' \in \mathbb{P}$. In this case t *goes* from s to s', and is an *outgoing transition* of s. States s and s' are the *source* and *target* of t. A *path* in an LTS is a finite or infinite alternating sequence of states and transitions, starting with a state, such that each transition goes from the state before it to the state after it (if any). A *completeness criterion* on an LTS is a set of its paths.

As for Kripke structures, a distributed system $\mathscr{D}$ can be modelled as a state s in an LTS upgraded with a completeness criterion. A (complete) run of $\mathscr{D}$ is then modelled by a complete path starting in s. As for Kripke structures, the default completeness criterion deems a path complete iff it either is infinite or ends in a *deadlock*, a state without outgoing transitions. An alternative completeness criterion could declare some infinite paths incomplete, saying that they do not model runs that can actually occur, and/or declare some finite paths that do not end in deadlock complete. A complete path π ending in a state models a run of the represented system that follows the path until its last state, and then stays in that state forever, without taking any of its outgoing transitions. A complete path that ends in a transition models a run in which the action represented by this last transition starts occurring but never finishes. It is often assumed that transitions are instantaneous, or at least of finite duration. This assumption is formalised through the adoption of a completeness criterion that holds all paths ending in a transition to be incomplete.

The most prominent translation from LTSs to Kripke structures stems from De Nicola & Vaandrager [13]. Its purpose is merely to efficiently lift the validity relation $\models$ from Kripke structures to LTSs. It simply creates a new state halfway along any transition labelled by a visible action, and moves the transition label to that state.

Definition 4.2. Let $(\mathbb{P}, Tr, src, trg, \ell)$ be an LTS over $Act = A \cup \{\tau\}$. The associated Kripke structure $(S, \to, \models)$ over A is given by

- $S := \mathbb{P} \dot\cup \{t \in Tr \mid \ell(t) \neq \tau\}$,
- $\to := \{(src(t), t), (t, trg(t)) \mid t \in Tr \wedge \ell(t) \neq \tau\} \cup \{(src(t), trg(t)) \mid t \in Tr \wedge \ell(t) = \tau\}$
- and $\models := \{(t, \ell(t)) \mid t \in Tr \wedge \ell(t) \neq \tau\}$.

Ignoring paths ending within a τ-transition, which are never deemed complete anyway, this translation yields a bijective correspondence between the paths in an LTS and those in its associated Kripke structure. Consequently, any completeness criterion on the LTS induces a completeness criterion on the Kripke structure. Hence it is now well-defined when $s \models^{CC} \varphi$, with s a state in an LTS, CC a completeness criterion on this LTS and φ an LTL formula.

4.2. Petri nets

Definition 4.3. A *(labelled) Petri net* over Act is a tuple $N = (S, T, F, M_0, \ell)$ where

- S and T are disjoint sets (of *places* and *transitions*),
- $F : (S \times T \cup T \times S) \to \mathbb{N}$ (the *flow relation*) such that $\forall t \in T. \exists s \in S.\ F(s,t) > 0$,
- $M_0 : S \to \mathbb{N}$ (the *initial marking*), and
- $\ell : T \to Act$ (the *labelling function*).

Petri nets are usually depicted by drawing the places as circles and the transitions as boxes, containing their label. For $x, y \in S \cup T$ there are $F(x,y)$ arrows (*arcs*) from x to y. When a Petri net represents a distributed system, a global state of this system is given as a *marking*, a multiset of places, depicted by placing $M(s)$ dots (*tokens*) in each place s. 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 *fires*. In that case, the transition t consumes $F(s,t)$ tokens from each place s. Naturally, this can happen only if M makes these tokens available in the first place. Next, the transition produces $F(t,s)$ tokens in each place s. Definition 4.5 formalises this notion of behaviour.

A *multiset* over a set X is a function $A : X \to \mathbb{N}$, i.e. $A \in \mathbb{N}^X$. Object $x \in X$ is an *element of* A iff $A(x) > 0$. The *empty* multiset, without elements, is denoted $\emptyset$. For multisets A and B over X I write $A \leq B$ iff $A(x) \leq B(x)$ for all $x \in X$; $A + B$ denotes the multiset over X with $(A+B)(x) := A(x) + B(x)$, $A \cap B$ is given by $(A \cap B)(x) := \min(A(x), B(x))$ and $A - B$ is given by $(A - B)(x) := A(x) \dot{-} B(x) = \max(A(x) - B(x), 0)$.

Definition 4.4. Let $N = (S, T, F, M_0, \ell)$ be a Petri net and $t \in T$. The multisets ${}^\bullet t,\ t^\bullet : S \to \mathbb{N}$ are given by ${}^\bullet t(s) = F(s,t)$ and $t^\bullet(s) = F(t,s)$ for all $s \in S$. The elements of ${}^\bullet t$ and $t^\bullet$ are called *pre-* and *postplaces* of t, respectively.

Definition 4.5. Let $N = (S, T, F, M_0, \ell)$ be a Petri net, $t \in T$ and $M, M' \in \mathbb{N}^S$. Transition t is *enabled* under M iff ${}^\bullet t \leq M$. In that case $M \xrightarrow{t}_N M'$, where $M' = (M - {}^\bullet t) + t^\bullet$.

A marking $M \in \mathbb{N}^S$ is *reachable* in a Petri net (S, T, F, M_0, ℓ) iff there are transitions $t_i \in T$ and markings $M_i \in \mathbb{N}^S$ for $i = 1, \ldots, k$, such that $M_k = M$ and $M_{i-1} \xrightarrow{t_i}_N M_i$ for $i = 1, \ldots, k$.

Definition 4.6 *([30]).* A Petri net $N = (S, T, F, M_0, \ell)$ is a *structural conflict net* if for all $t, u \in T$ with ${}^\bullet t \cap {}^\bullet u \neq \emptyset$ and for all reachable markings M, one has ${}^\bullet t + {}^\bullet u \not\leq M$.

Here I restrict myself to structural conflict nets, henceforth simply called *nets*, a class of Petri nets containing the *safe* Petri nets that are normally used to give semantics to process algebras.

Definition 4.7. Given a net $N = (S, T, F, M_0, \ell)$, its associated LTS $(\mathbb{P}, Tr, src, trg, \ell)$ is given by $\mathbb{P} := \mathbb{N}^S$, $Tr := \{(M,t) \in \mathbb{N}^S \times T \mid {}^\bullet t \leq M\}$, $src(M,t) := M$, $trg(M,t) := (M - {}^\bullet t) + t^\bullet$ and $\ell(M,t) := \ell(t)$. The net N maps to the state M_0 in this LTS.

A *completeness criterion* on a net is a completeness criterion on its associated LTS. Now $N \models^{CC} \varphi$ is defined to hold iff $M_0 \models^{CC} \varphi$ in the associated LTS.

4.3. CCS

The *Calculus of Communicating Systems* (CCS) [47] is parametrised with sets $\mathscr{K}$ of *agent identifiers* and $\mathscr{A}$ of *names*; each $X \in \mathscr{K}$ comes with a defining equation $X \stackrel{def}{=} P$ with P being a CCS expression as defined below. $Act := \mathscr{A} \dot\cup \bar{\mathscr{A}} \dot\cup \{\tau\}$ is the set of *actions*, where τ is a special *internal action* and $\bar{\mathscr{A}} := \{\bar a \mid a \in \mathscr{A}\}$ is the set of *co-names*. Complementation is extended to $\bar{\mathscr{A}}$ by setting $\bar{\bar a} = a$. Below, $\underline{a}$ ranges over $\mathscr{A} \cup \bar{\mathscr{A}}$, α over Act, and X, Y over $\mathscr{K}$. A *relabelling* is a function $f : \mathscr{A} \to \mathscr{A}$; it extends to Act by $f(\bar a) = \overline{f(a)}$ and $f(\tau) := \tau$. The set $\mathrm{T_{CCS}}$ of CCS expressions or *processes* is the smallest set including:

$\sum_{i\in I} \alpha_i.P_i$	for I an index set, $\alpha_i \in Act$ and $P_i \in \mathrm{T_{CCS}}$	*guarded choice*
$P \mid Q$	for $P, Q \in \mathrm{T_{CCS}}$	*parallel composition*
$P \backslash L$	for $L \subseteq \mathscr{A}$ and $P \in \mathrm{T_{CCS}}$	*restriction*
$P[f]$	for f a relabelling and $P \in \mathrm{T_{CCS}}$	*relabelling*
X	for $X \in \mathscr{K}$	*agent identifier*

The process $\sum_{i\in\{1,2\}} \alpha_i.P_i$ is often written as $\alpha_1.P_1 + \alpha_2.P_2$, $\sum_{i\in\{1\}} \alpha_i.P_i$ as $\alpha_1.P_1$ and $\sum_{i\in\emptyset} \alpha_i.P_i$ as $\mathbf{0}$. The semantics of CCS is given by the transition relation $\to \subseteq \mathrm{T_{CCS}} \times Act \times \mathscr{P}(\mathscr{C}) \times \mathrm{T_{CCS}}$, where transitions $P \xrightarrow{\alpha, C} Q$ are derived from the rules of Table 1. Ignoring the labels $C \in \mathscr{P}(\mathscr{C})$ for now, such a transition indicates that process P can perform the action $\alpha \in Act$

Table 1
Structural operational semantics of CCS.

$$\sum_{i\in I}\alpha_i.P_i \xrightarrow{\alpha_j\,,\,\{\varepsilon\}} P_j \quad (j\in I)$$

$$\frac{P \xrightarrow{\alpha\,,\,C} P'}{P|Q \xrightarrow{\alpha\,,\,\mathrm{L}\cdot C} P'|Q} \qquad \frac{P \xrightarrow{a\,,\,C} P',\ Q \xrightarrow{\bar a\,,\,D} Q'}{P|Q \xrightarrow{\tau\,,\,\mathrm{L}\cdot C\cup \mathrm{R}\cdot D} P'|Q'} \qquad \frac{Q \xrightarrow{\alpha\,,\,D} Q'}{P|Q \xrightarrow{\alpha\,,\,\mathrm{R}\cdot D} P|Q'}$$

$$\frac{P \xrightarrow{\alpha\,,\,C} P'}{P\backslash L \xrightarrow{\alpha\,,\,C} P'\backslash L}\ (\alpha,\bar\alpha\notin L) \qquad \frac{P \xrightarrow{\alpha\,,\,C} P'}{P[f] \xrightarrow{f(\alpha)\,,\,C} P'[f]} \qquad \frac{P \xrightarrow{\alpha\,,\,C} P'}{X \xrightarrow{\alpha\,,\,C} P'}\ (X \stackrel{def}{=} P)$$

and transform into process Q. The process $\sum_{i\in I}\alpha_i.P_i$ performs one of the actions α_j for $j\in I$ and subsequently acts as P_j. The parallel composition $P|Q$ executes an action from P, an action from Q, or a synchronisation between complementary actions c and $\bar c$ performed by P and Q, resulting in an internal action τ. The restriction operator $P\backslash L$ inhibits execution of the actions from L and their complements. The relabelling $P[f]$ acts like process P with all labels α replaced by $f(\alpha)$. Finally, the rule for agent identifiers says that an agent X has the same transitions as the body P of its defining equation. The standard version of CCS [47] features a *choice* operator $\sum_{i\in I}P_i$; here I use the fragment of CCS that merely features guarded choice.

The second label of a transition indicates the set of (parallel) *components* involved in executing this transition. The set $\mathscr{C}$ of components is defined as $\{\mathrm{L},\mathrm{R}\}^*$, that is, the set of strings over the indicators Left and Right, with $\varepsilon\in\mathscr{C}$ denoting the empty string and $\mathrm{D}\cdot C := \{\mathrm{D}\sigma \mid \sigma\in C\}$ for $\mathrm{D}\in\{\mathrm{L},\mathrm{R}\}$ and $C\subseteq\mathscr{C}$.

Example 4.8. The process $P := (X|\bar a.\mathbf{0})|\bar a.b.\mathbf{0}$ with $X \stackrel{def}{=} a.X$ has as outgoing transitions $P \xrightarrow{a\,,\,\{\mathrm{LL}\}} P$, $P \xrightarrow{\tau\,,\,\{\mathrm{LL},\mathrm{LR}\}} (X|\mathbf{0})|\bar a.b.\mathbf{0}$, $P \xrightarrow{\bar a\,,\,\{\mathrm{LR}\}} (X|\mathbf{0})|\bar a.b.\mathbf{0}$, $P \xrightarrow{\tau\,,\,\{\mathrm{LL},\mathrm{R}\}} (X|\bar a.\mathbf{0})|b.\mathbf{0}$ and $P \xrightarrow{\bar a\,,\,\{\mathrm{R}\}} (X|\bar a.\mathbf{0})|b.\mathbf{0}$.

These components stem from Victor Dyseryn [personal communication, 2017] and were introduced in [26]. They were not part of the standard semantics of CCS [47], which can be retrieved by ignoring them.

Definition 4.9. The LTS of CCS is $(\mathrm{T}_{\mathrm{CCS}}, Tr, src, trg, \ell)$, with Tr the set of derivable transitions $P \xrightarrow{\alpha\,,\,C} Q$, $\ell(P \xrightarrow{\alpha\,,\,C} Q) = \alpha$, $src(P \xrightarrow{\alpha\,,\,C} Q) = P$ and $trg(P \xrightarrow{\alpha\,,\,C} Q) = Q$. Employing this interpretation of CCS, one can pronounce judgements $P \models^{CC} \varphi$ for CCS processes P.

4.4. Labelled transition systems with concurrency

Definition 4.10. An *LTS with concurrency* (LTSC) is a tuple $(\mathbb{P}, Tr, src, trg, \ell, \frown)$ consisting of a LTS $(\mathbb{P}, Tr, src, trg, \ell)$ and a *concurrency relation* $\frown \subseteq Tr\times Tr$, such that:

$$t \not\frown t \text{ for all } t\in Tr, \tag{4.1}$$

$$\text{if } t\in Tr \text{ and } \pi \text{ is a path from } src(t) \text{ to } s\in\mathbb{P} \text{ such that } t\frown v \text{ for all transitions } v \text{ occurring in } \pi, \text{ then there is a } u\in Tr \text{ such that } src(u)=s,\ \ell(u)=\ell(t) \text{ and } t\not\frown u. \tag{4.2}$$

Informally, $t\frown v$ means that the transition v does not interfere with t, in the sense that it does not affect any resources that are needed by t, so that in a state where t and v are both possible, after doing v one can still do a future variant u of t. Write $t\smile v$ for $t\frown v \wedge v\frown t$.

LTSCs were introduced in [25], although there the model is more general on various counts; I do not need this generality here.

The LTS associated with CCS can be turned into an LTSC by defining $(P \xrightarrow{\alpha\,,\,C} P') \frown (Q \xrightarrow{\beta\,,\,D} Q')$ iff $C\cap D=\emptyset$, that is, two transitions are concurrent iff they stem from disjoint sets of components [33,26]. In this LTSC, and many others, including the ones associated to nets below, $\frown$ is symmetric, and thus the same as $\smile$.

Example 4.11. Let the 5 transitions from Example 4.8 be t, u, v, w and x, respectively. Then $t\not\smile w$ because these transitions share the component LL. Yet $v\smile w$.

The LTS associated with a net can be turned into an LTSC by defining $(M,t)\frown(M',u)$ iff ${}^\bullet t\cap {}^\bullet u=\emptyset$, i.e., the two LTS-transitions stem from net-transitions that have no preplaces in common. Naturally, any LTSC can be turned into a LTS, and further into a Kripke structure, by forgetting $\frown$.

5. Progress, justness and fairness

With the above definitions one can pronounce judgements $\mathscr{D} \models^{CC} \varphi$ for distributed systems $\mathscr{D}$ given as a net or a CCS expression, for instance. Through the translations of Definitions 4.7 or 4.9 one renders $\mathscr{D}$ as a state P in an LTS. The completeness criterion CC is given as a set of paths on that LTS. Then, using Definition 4.2, P is seen as a state in a Kripke structure, and CC as a set of paths on that Kripke structure. Here it is well-defined when $P \models^{CC} \varphi$ holds, and this verdict applies to the judgement $\mathscr{D} \models^{CC} \varphi$.

The one thing left to explain is where the completeness criterion CC comes from. In this section I define completeness criteria $CC \in \{SF(\mathscr{T}), WF(\mathscr{T}), J, Pr, \top \mid \mathscr{T} \in \mathscr{P}(\mathscr{P}(Tr))\}$ on LTSs $(\mathbb{P}, Tr, src, trg, \ell)$, to be used in judgements $P \models^{CC} \varphi$, for $P \in \mathbb{P}$ and φ an LTL formula. These criteria are called *strong fairness* (*SF*), *weak fairness* (*WF*), both parametrised with a set $\mathscr{T} \subseteq \mathscr{P}(Tr)$ of *tasks*, *justness* (J), *progress* (*Pr*) and the *trivial* completeness criterion ($\top$). Justness is merely defined on LTSCs. I confine myself to criteria that hold finite paths ending within a transition to be incomplete.

Reading Example 3.3, one could find it unfair that Bart might never get a beer. Strong and weak *fairness* are completeness criteria that postulate that Bart will get a beer, namely by ruling out as incomplete the infinite paths in which he does not. They are formalised by introducing a set $\mathscr{T}$ of *tasks*, each being a set of transitions (in an LTS or Kripke structure).

Definition 5.1 *([33]).* A task $T \in \mathscr{T}$ is *enabled* in a state s iff s has an outgoing transition from T. It is *perpetually enabled* on a path π iff it is enabled in every state of π. It is *relentlessly enabled* on π, if each suffix of π contains a state in which it is enabled.[7] It *occurs* in π if π contains a transition $t \in T$.

A path π is *weakly fair* if, for every suffix π' of π, each task that is perpetually enabled on π', occurs in π'. It is *strongly fair* if, for every suffix π' of π, each task that is relentlessly enabled on π', occurs in π'.

As completeness criteria, these notions take only the fair paths to be complete. In Example 3.3 it suffices to have a task "Bart gets a beer", consisting of the three transitions leading to the B state. Now in any path in which Bart never gets a beer this task is perpetually enabled, yet never taken. Hence weak fairness suffices to rule out such paths. One has $\mathscr{D} \models^{WF(\mathscr{T})} \mathbf{F}B$.

Local fairness [33] allows the tasks $\mathscr{T}$ to be declared on an ad hoc basis for the application at hand. On this basis one can call it unfair if Bart doesn't get a beer, without requiring that Cameron should get a beer as well. *Global fairness*, on the other hand, distils the tasks of an LTS in a systematic way out of the structure of a formalism, such as pseudocode, process algebra or Petri nets, that gave rise to the LTS. A classification of many ways to do this, and thus of many notions of strong and weak fairness, appears in [33]. In *fairness of directions* [20], for instance, each transition in an LTS is assumed to stem from a particular *direction*, or *instruction*, in the pseudocode that generated the LTS; now each direction represents a task, consisting of all transitions derived from that direction.

In [33] the assumption that a system will never stop when there are transitions to proceed is called *progress*. In Example 3.4 it takes a progress assumption to conclude that Bart will get his beer. Progress fits the default completeness criterion introduced before, i.e., $\models^{Pr}$ is the same as $\models$. Not (even) assuming progress can be formalised by the trivial completeness criterion $\top$ that declares all paths to be complete. Naturally, $\mathscr{E} \not\models^{\top} \mathbf{F}B$.

Completeness criterion D is called *stronger* than criterion C if it rules out more paths as incomplete. So $\top$ is the weakest of all criteria, and, for any given collection $\mathscr{T}$, strong fairness is stronger than weak fairness. When assuming that each transition occurs in at least one task—which can be ensured by incorporating a default task consisting of all transitions—progress is weaker than weak fairness.

Justness [33] is a strong form of progress, defined on LTSCs.

Definition 5.2. A path π is *just* if for each transition t whose source state $s := src(t)$ occurs in π, the (or any) suffix of π starting at s contains a transition u with $t \not\frown u$.

Example 5.3. The infinite path π that only ever takes transition t in Example 4.8/4.11 is unjust. Namely with transition v in the rôle of the t from Definition 5.2, π contains no transition y with $v \not\frown y$.

Informally, the only reason for an enabled transition not to occur, is that one of its resources is eventually used for some other transition. In Example 3.5 for instance, the orders of Alice and Cameron are clearly concurrent with the one of Bart, in the sense that they do not compete for shared resources. Taking t to be the transition in which Bart gets his beer, any path in which t does not occur is unjust. Thus $\mathscr{F} \models^{J} \mathbf{F}B$.

For most choices of $\mathscr{T}$ found in the literature, weak fairness is a strictly stronger completeness criterion than justness. In Example 3.3, for instance, the path in which Bart does not get a beer is just. Namely, any transition u giving Alice or Cameron a beer competes for the same resource as the transition t giving Bart a beer, namely the attention of the bartender. Thus $t \not\frown u$, and consequently $\mathscr{D} \not\models^{J} \mathbf{F}B$.

[7] This is the case if the task is enabled in infinitely many states of π, in a state that occurs infinitely often in π, or in the last state of a finite π.

6. Blocking actions

I now present *reactive* temporal logic by extending the ternary judgements $P \models^{CC} \varphi$ defined above to quaternary judgements $P \models_B^{CC} \varphi$, with $B \subseteq A$ a set of *blocking* actions. Here A is the set of all observable actions of the LTS on which LTL is interpreted. The intuition is that actions $b \in B$ may be blocked by the environment, but actions $a \in A \setminus B$ may not. The relation $\models_B$ can be used to formalise the assumption that the actions in $A \setminus B$ are not under the control of the user of the modelled system, or that there is an agreement with the user not to block them. Either way, it is a disclaimer on the wrapping of our temporal judgement, that it is valid merely when applying the involved distributed system in an environment that may block actions from B only. The hidden action τ may never be blocked.

I will present the relations $\models_B^{CC}$ for each choice of $CC \neq \top$ discussed in the previous section, and each $B \subseteq A$. When writing $P \models_B^{CC} \varphi$ the modifier B adapts the default completeness criterion by declaring certain finite paths complete, and the modifier $CC \neq \top, Pr$ adapts it by declaring some infinite paths incomplete.

Starting with $CC = Pr$, I call a path *B-progressing* iff it is either infinite or ends in a state of which all outgoing transitions have a label from B, and write $s \models_B^{Pr} \varphi$, or $s \models_B \varphi$ for short, if $\pi \models \varphi$ holds for all B-progressing paths π starting in s. The completeness criterion *B-progress*, which takes the B-progressing to be the complete ones, says that a system may stop in a state with outgoing transitions only when they are all blocked by the environment. Note that the standard LTL interpretation $\models$ is simply $\models_\emptyset$, obtained by taking the empty set of blocking actions.

In the presence of the modifier B, Definition 5.2 is adapted as follows:

Definition 6.1. A path π is *B-just* if for each $t \in Tr$ with $\ell(t) \notin B$ and whose source state $s := src(t)$ occurs in π, any suffix of π starting at s contains a transition u with $t \not\frown u$.

It doesn't matter whether $\ell(u) \in B$. The completeness criterion *B-justness* takes the B-just paths to be the complete ones. Write $s \models_B^J \varphi$ if $\pi \models \varphi$ for all B-just paths π starting in s.

For the remaining cases $CC = SF(\mathscr{T})$ and $CC = WF(\mathscr{T})$, adapt the first sentence of Definition 5.1 as follows.

Definition 6.2. A task $T \in \mathscr{T}$ is *B-enabled* in a state s iff s has an outgoing transition $t \in T$ with $\ell(t) \notin B$.

Strong and weak B-fairness of paths is then defined as in Definition 5.1, but replacing "enabled" by "B-enabled".

The above completes the formal definition of the validity of temporal judgements $P \models_B^{CC} \varphi$ with φ an LTL formula, $B \subseteq A$, and either

- $CC = Pr$ and P a state in an LTS, a Petri net or a CCS expression,
- $CC = J$ and P a state in an LTSC, a Petri net or a CCS expression,
- $CC = WF(\mathscr{T})$ or $SF(\mathscr{T})$ and P a state in an LTS $(\mathbb{P}, Tr, src, trg, \ell)$ with $\mathscr{T} \in \mathscr{P}(\mathscr{P}(Tr))$, or P a net or CCS expression with associated LTS $(\mathbb{P}, Tr, src, trg, \ell)$ and $\mathscr{T} \in \mathscr{P}(\mathscr{P}(Tr))$.

Namely, in case P is a state in an LTS, it is also a state in the associated Kripke structure K. Moreover, B and CC combine into a single completeness criterion $CC(B)$ on that LTS, which translates as a completeness criterion $CC(B)$ on K. Now Definition 3.2 tells whether $P \models^{CC(B)} \varphi$ holds.

In case $CC = J$ and P a state in an LTSC, B and J combine into a single completeness criterion $J(B)$ on that LTSC, which is also a completeness criterion on the associated LTS; now proceed as above.

In case P is a Petri net or CCS expression, first translate it into a state in an LTS or LTSC, using Definitions 4.7 or 4.9, respectively, and proceed as above.

Temporal judgements $P \models_B^{CC} \varphi$, as introduced above, are not limited to the case that φ is an LTL formula. In [28] I show that allowing φ to be a CTL formula instead poses no additional complications, and I expect the same to hold for other temporal logics.

Judgements $P \models_B^{CC} \varphi$ get stronger (= less likely true) when the completeness criterion CC is weaker, and the set B of blocking actions larger.

7. Translating reactive LTL into standard LTL

Here I translate judgements $s \models_B^{CC} \varphi$ in reactive LTL[8] into equivalent judgements $\hat{s} \models \psi$ in traditional LTL, albeit with infinite conjunctions. The price to be paid for this is an extra dose of atomic propositions. I start with judgements $s \models_B^{CC} \varphi$ interpreted in an LTS $(\mathbb{P}, Tr, src, trg, \ell)$, as this is where the completeness criterion $CC(B)$ takes shape. I use a slightly different translation from LTSs to Kripke structures than the one of Definition 4.2; it inserts a state halfway along *any* transition, even if it is labelled τ. However, τ will not be an atomic proposition of the resulting Kripke structure K, and the

[8] *Reactive LTL* refers to judgements of the form $s \models_B^{CC} \varphi$ or $\mathscr{D} \models_B^{CC} \varphi$ where φ is an LTL formula.

new halfway states do not inherit a transition label. This change affects the bookkeeping for next-state operators, but not in a bad way.

To translate $\models_B^{CC}$ into $\models$, I have to make provisions for finite $CC(B)$-complete paths that are $Pr(\emptyset)$-incomplete, and for infinite $CC(B)$-incomplete paths that are $Pr(\emptyset)$-complete. In order not to tackle these opposite forces in the same step, I first present a translation from reactive LTL into LTL with the trivial completeness criterion. That is, for each choice of CC from Section 5, each set $B \subseteq A$ of blocking actions, and each LTL formula φ, I present an LTL formula φ_B^{CC} such that $s \models_B^{CC} \varphi$ iff $s \models^\top \varphi_B^{CC}$ for each $s \in \mathbb{P}$.

Given a collection $\mathscr{T}$ of tasks and a set B of blocking actions, introduce for each task $T \in \mathscr{T}$ two atomic propositions en_B^T and oc^T. Proposition en_B^T holds in those states of K that stem from states of $s \in \mathbb{P}$ in which the task T is B-enabled, i.e., that have an outgoing transition $t \in T$ with $\ell(t) \notin B$. Additionally, it holds in those states of K that stem from transitions $t \in Tr$ such that T is B-enabled in both $src(t)$ and $trg(t)$. Proposition oc^T holds in those states of K that stem from a transition $t \in T$; this is where the task *occurs*. Now the formula

$$WF(\mathscr{T})_B := \bigwedge_{T \in \mathscr{T}} \mathbf{G}(\mathbf{G}en_B^T \Rightarrow \mathbf{F}oc^T)$$

holds for a path π of K exactly when π is weakly B-fair. Hence the formula $WF(\mathscr{T})_B \Rightarrow \varphi$ says that φ holds on weakly B-fair paths, and one has $s \models_B^{WF(\mathscr{T})} \varphi$ iff $s \models^\top WF(\mathscr{T})_B \Rightarrow \varphi$. Likewise,

$$SF(\mathscr{T})_B := \bigwedge_{T \in \mathscr{T}} \mathbf{G}(\mathbf{GF}en_B^T \Rightarrow \mathbf{F}oc^T)$$

holds for a path of K iff it is strongly B-fair, so that $s \models_B^{SF(\mathscr{T})} \varphi$ iff $s \models^\top SF(\mathscr{T})_B \Rightarrow \varphi$. The formulas $\mathbf{G}(\mathbf{G}en \Rightarrow \mathbf{F}oc)$ and $\mathbf{G}(\mathbf{GF}en \Rightarrow \mathbf{F}oc)$ stem from [21]. In the literature one sometimes find the equivalent forms $\mathbf{FG}en \Rightarrow \mathbf{GF}oc$ and $\mathbf{GF}en \Rightarrow \mathbf{GF}oc$.

Progress, i.e., the case $CC = Pr$, can be dealt with in the same way, by recognising it as weak or strong fairness involving a single task, spanning all transitions.

To deal with B-justness, introduce atomic propositions en^t and $\sharp t$ for each transition $t \in Tr$ with $\ell(t) \notin B$. Proposition en^t holds in the state $src(t)$ only, whereas $\sharp t$ holds in all states of K that stem from a transition $u \in Tr$ with $t \not\rightsquigarrow u$. Now

$$J_B := \bigwedge_{t \in Tr,\ \ell(t) \notin B} \mathbf{G}(en^t \Rightarrow \mathbf{F}\sharp t)$$

holds for a path of K iff it is B-just. Consequently, $s \models_B^J \varphi$ iff $s \models^\top J_B \Rightarrow \varphi$.

It remains to translate $\models^\top$ into $\models$. To this end, I transform K into $\widehat{K}$ by adding a self-loop at each of its states. I also introduce a fresh atomic proposition tr, for *transition*, and use it to label all states in $\widehat{K}$ that stem from a transition from Tr. For each finite path π in K let π^∞ be the infinite path in $\widehat{K}$ obtained by repeating the last state of π infinitely often. In case π is infinite, let $\pi^\infty := \pi$. Let Z be the completeness criterion on $\widehat{K}$ that declares each path of the form π^∞ complete. These are exactly the infinite paths without a subsequence sst with $s \neq t$. So $\_^\infty$ is a bijection between the paths of K and the Z-complete paths of $\widehat{K}$. Let $\mathscr{Q}$ be the transformation on LTL formula, defined by

$$\begin{array}{lllll}
\mathscr{Q}(p) := p & & \mathscr{Q}(\neg\varphi) := \neg\mathscr{Q}(\varphi) & & \mathscr{Q}(\varphi \wedge \psi) := \mathscr{Q}(\varphi) \wedge \mathscr{Q}(\psi) \\
\mathscr{Q}(\mathbf{F}\varphi) := \mathbf{F}\mathscr{Q}(\varphi) & & \mathscr{Q}(\mathbf{G}\varphi) := \mathbf{G}\mathscr{Q}(\varphi) & & \mathscr{Q}(\varphi\mathbf{U}\psi) := \mathscr{Q}(\varphi)\mathbf{U}\mathscr{Q}(\psi) \\
\multicolumn{5}{l}{\mathscr{Q}(\mathbf{X}\varphi) := \big(tr \Rightarrow \mathbf{X}(\neg tr \wedge \mathscr{Q}(\varphi))\big) \wedge \big(\neg tr \Rightarrow \mathbf{X}(tr \wedge \mathscr{Q}(\varphi))\big).}
\end{array}$$

A trivial induction on φ shows that $\pi \models \varphi$ holds in K iff $\pi^\infty \models \mathscr{Q}(\varphi)$ holds in $\widehat{K}$. This implies that $s \models^\top \varphi$ holds in K iff $s \models^Z \varphi$ holds in $\widehat{K}$; I will write the latter as $\hat{s} \models^Z \varphi$. Z-completeness can be stated as

$$\mathscr{Z} := \mathbf{G}\big(tr \Rightarrow (\mathbf{G}tr \vee \mathbf{X}(\neg tr))\big) \wedge \mathbf{G}\big(\neg tr \Rightarrow (\mathbf{G}(\neg tr) \vee \mathbf{X}tr)\big).$$

Hence $s \models^\top \varphi$ iff $\hat{s} \models \mathscr{Z} \Rightarrow \varphi$.

The above translation from reactive LTL into standard LTL may give the impression that reactive LTL is not more expressive than standard LTL. This conclusion is not really valid, due to the addition to the formalism of fresh atomic propositions. It is for instance widely accepted that LTL is not more expressive than CTL. Yet, if one introduces an atomic proposition p_φ for each CTL formula φ, one that is declared to hold for all states that satisfy φ, one trivially obtains $s \models \varphi$ iff $s \models p_\varphi$. This would suggest that CTL can be faithfully translated into LTL, even without using any of the modal operators of LTL.

8. Safety properties

A *safety property* is a temporal formula φ that holds for a path π iff it holds for all finite prefixes of π. In that case $\mathscr{D} \models \varphi$ iff $\pi \models \varphi$ for all finite paths π starting in the initial state of $\mathscr{D}$. Such a property can be thought to say that nothing bad will ever happen [43]. The intuition is that a bad thing must be observable in a finite prefix of a run, so that $\mathscr{D} \models \neg\varphi$ iff $\pi \models \neg\varphi$ for some finite path π starting in the initial state of $\mathscr{D}$.

Proposition 8.1. *The fragment of LTL given by the grammar*

$$\varphi, \psi ::= p \mid \neg p \mid \varphi \wedge \psi \mid \varphi \vee \psi \mid \mathbf{Y}\varphi \mid \mathbf{G}\varphi \mid \psi \mathbf{W} \varphi$$

describes only safety properties. Here $\mathbf{Y}$ *is the dual of* $\mathbf{X}$*, and* $\psi \mathbf{W} \varphi$ *abbreviates* $(\psi \mathbf{U} \varphi) \vee \mathbf{G}\psi$.

Proof. Let φ and ψ be safety properties. I show that also $\varphi \vee \psi$ is a safety property.

Let $\pi \models \varphi \vee \psi$. Then either $\pi \models \varphi$ or $\pi \models \psi$; for reasons of symmetry I may assume the former. Let π' be a finite prefix of π. Since φ is a safety property, $\pi' \models \varphi$. Hence $\pi' \models \varphi \vee \psi$, which needed to be shown.

Now let π be a path such that $\pi' \models \varphi \vee \psi$ for all finite prefixes π' of π. In case $\pi' \models \varphi$ for all finite prefixes π' of π, it follows that $\pi \models \varphi$, since φ is a safety property, and hence $\pi \models \varphi \vee \psi$. So assume π has a finite prefix π'' for which $\pi'' \not\models \varphi$. Since φ is a safety property, it follows that $\pi' \not\models \varphi$ for each finite path π' with $\pi'' \leq \pi' \leq \pi$. So $\pi' \models \psi$ for all such π'. As $\pi'' \models \psi$, and ψ is a safety property, one also has $\pi' \models \psi$ for all prefixes π' of π''. Thus $\pi' \models \psi$ for *all* prefixes π' of π. As ψ is a safety property, it follows that $\pi \models \psi$, so $\pi \models \varphi \vee \psi$.

That p and $\neg p$ are safety properties, for $p \in AP$, and that the safety properties are closed under conjunction, is trivial.

Let φ be a safety property. I show that $\mathbf{Y}\varphi$ and $\mathbf{G}\varphi$ are safety properties.

Let $\pi \models \mathbf{Y}\varphi$. Then either $|\pi| = 0$ or $\pi_{+1} \models \varphi$. Let π' be a finite prefix of π. Then either $|\pi'| = 0$, so that trivially $\pi' \models \mathbf{Y}\varphi$, or π'_{+1} is a finite prefix of π_{+1}. Since φ is a safety property, $\pi'_{+1} \models \varphi$, and thus $\pi' \models \mathbf{Y}\varphi$.

Now let π be a path such that $\pi' \models \mathbf{Y}\varphi$ for all finite prefixes π' of π. In case $|\pi| = 0$, trivially $\pi \models \mathbf{Y}\varphi$. So assume $|\pi| > 0$. Then $\pi'_{+1} \models \varphi$ for all finite prefixes π' of π with $|\pi'| > 0$, that is, $\rho \models \varphi$ for all finite prefixes ρ of π_{+1}. Since φ is a safety property it follows that $\pi_{+1} \models \varphi$, and hence $\pi \models \mathbf{Y}\varphi$.

Let $\pi \models \mathbf{G}\varphi$. Then $\pi \restriction \rho \models \varphi$ for each finite prefix ρ of π. As φ is a safety property, $\rho' \models \varphi$ for each finite prefix ρ' of $\pi \restriction \rho$, for each finite prefix ρ of π, that is, $\pi' \restriction \rho \models \varphi$ for each pair of finite prefixes (ρ, π') with $\rho \leq \pi' \leq \pi$. Thus $\pi' \models \mathbf{G}\varphi$ for each finite prefix π' of π.

Now let π be a path such that $\pi' \models \mathbf{G}\varphi$ for all finite prefixes π' of π. Then $\pi' \restriction \rho \models \varphi$ for each pair of finite prefixes (ρ, π') with $\rho \leq \pi' \leq \pi$, that is, $\rho' \models \varphi$ for each finite prefix ρ' of $\pi \restriction \rho$, for each finite prefix ρ of π. As φ is a safety property, $\pi \restriction \rho \models \varphi$ for each finite prefix ρ of π. Thus $\pi \models \mathbf{G}\varphi$.

Finally, let φ and ψ be safety properties. I show that $\psi \mathbf{W} \varphi$ is a safety property.

Let $\pi \models \psi \mathbf{W} \varphi = (\psi \mathbf{U} \varphi) \vee \mathbf{G}\psi$. One possibility is that $\pi \models \mathbf{G}\psi$. Then, as shown above, for each finite prefix π' of π one has $\pi' \models \mathbf{G}\psi$, and thus $\pi' \models \psi \mathbf{W} \varphi$. The other possibility is that $\pi \models \psi \mathbf{U} \varphi$. Then there is a finite prefix ρ of π such that $\pi \restriction \rho \models \varphi$, and for each $\zeta < \rho$ one has $\pi \restriction \zeta \models \psi$. Now let π' be a finite prefix of π.

First assume that $\pi' < \rho$. Then for each $\zeta \leq \pi'$ the path $\pi' \restriction \zeta$ is a finite prefix of $\pi \restriction \zeta$. Since ψ is a safety property, this implies $\pi' \restriction \zeta \models \psi$. Thus $\pi' \models \mathbf{G}\psi$ and hence $\pi' \models \psi \mathbf{W} \varphi$.

Next assume that $\rho \leq \pi'$. Then $\pi' \restriction \rho$ is a finite prefix of $\pi \restriction \rho$. Since φ is a safety property, $\pi' \restriction \rho \models \varphi$. For each $\zeta < \rho$, $\pi' \restriction \zeta$ is a finite prefix of $\pi \restriction \zeta$. Thus $\pi' \restriction \zeta \models \psi$, as ψ is a safety property. It follows that $\pi' \models \psi \mathbf{U} \varphi$ and hence $\pi' \models \psi \mathbf{W} \varphi$.

Now let $\pi \not\models \psi \mathbf{W} \varphi$. Then $\pi \not\models \mathbf{G}\psi$, so there is a finite prefix ρ of π with $\pi \restriction \rho \not\models \psi$, and, choosing ρ as short as possible, $\pi \restriction \zeta \models \psi$ for all $\zeta < \rho$. Since $\pi \not\models \psi \mathbf{U} \varphi$, one has $\pi \restriction \zeta \not\models \varphi$ for each $\zeta \leq \rho$.[9] As φ is a safety property, for each $\zeta \leq \rho$ there exists a finite prefix ζ' of $\pi \restriction \zeta$ with $\zeta' \not\models \varphi$; or in other words, for each $\zeta \leq \rho$ there is a finite $\zeta \leq \pi_\zeta \leq \pi$ with $\pi_\zeta \restriction \zeta \not\models \varphi$. As ψ is a safety property, there exists a finite prefix ζ' of $\pi \restriction \rho$ with $\zeta' \not\models \psi$; or in other words, a finite $\rho \leq \bar{\pi} \leq \pi$ with $\bar{\pi} \restriction \rho \not\models \psi$. Now let $\pi' \leq \pi$ be the longest of the finite paths $\bar{\pi}$ and π_ζ for each $\zeta \leq \rho$. Since ψ and φ and safety properties, $\pi' \restriction \rho \not\models \psi$, and $\pi' \restriction \zeta \not\models \psi$ for each $\zeta \leq \rho$. It follows that $\pi' \not\models \psi \mathbf{W} \varphi$. □

For safety properties φ, the reactive part of reactive temporal logic is irrelevant, as one has $\mathscr{D} \models^{CC}_{B} \varphi$ iff $\mathscr{D} \models \varphi$, for all completeness criteria CC and all $B \subseteq A$. Namely, both hold iff $\pi \models \varphi$ for all finite paths π starting in the initial state of $\mathscr{D}$. This hinges on the requirement of *feasibility* imposed on completeness criteria [1,33]: any finite partial run can be extended to a complete run; or any finite path must be a prefix of a complete path.

Part II. Formalising mutual exclusion and fair scheduling in reactive LTL

Here I recall the mutual exclusion problem as posed by Dijkstra [16], and the related notion of a fair scheduler [31]. Employing reactive LTL, I formulate requirements that tell exactly what does and does not count as a mutual exclusion protocol, and as a fair scheduler. Since my requirements are parametrised by completeness criteria, which are progress, justness or fairness assumptions, I obtain a hierarchy of quality criteria for mutual exclusion protocols and fair schedulers, where a weaker completeness criterion characterises a higher quality protocol. When allowing (strong or) weak fairness as parameter in my requirements, an intuitively unsatisfactory mutual exclusion protocol or fair scheduler, which I call the *gatekeeper*, meets all requirements. This indicates that weak fairness is too strong an assumption to be used in these parameters.

[9] This argument shows that $\mathbf{U}$ and $\mathbf{W}$ are almost duals: $\neg(\psi \mathbf{W} \varphi) \equiv (\neg\varphi)\mathbf{U}(\neg\psi \wedge \neg\varphi)$, and thus, using that $\psi \mathbf{U} \varphi \equiv (\psi \mathbf{W} \varphi) \wedge \mathbf{F}\varphi$, also $\neg(\psi \mathbf{U} \varphi) \equiv (\neg\varphi)\mathbf{W}(\neg\psi \wedge \neg\varphi)$.

9. The mutual exclusion problem and its history

The mutual exclusion problem was presented by Dijkstra in [16] and formulated as follows:

> "To begin, consider N computers, each engaged in a process which, for our aims, can be regarded as cyclic. In each of the cycles a so-called "critical section" occurs and the computers have to be programmed in such a way that at any moment only one of these N cyclic processes is in its critical section. In order to effectuate this mutual exclusion of critical-section execution the computers can communicate with each other via a common store. Writing a word into or nondestructively reading a word from this store are undividable operations; i.e., when two or more computers try to communicate (either for reading or for writing) simultaneously with the same common location, these communications will take place one after the other, but in an unknown order."

Dijkstra proceeds to formulate a number of requirements that a solution to this problem must satisfy, and then presents a solution that satisfies those requirements. The most central of these are:

- (*Mutex*) "no two computers can be in their critical section simultaneously", and
- (*Deadlock-freedom*) if at least one computer intends to enter its critical section, then at least one "will be allowed to enter its critical section in due time".

Two other important requirements formulated by Dijkstra are

- (*Speed independence*) "Nothing may be assumed about the relative speeds of the N computers",
- and (*Optionality*) "If any of the computers is stopped well outside its critical section, this is not allowed to lead to potential blocking of the others."

A crucial assumption is that each computer, in each cycle, spends only a finite amount of time in its critical section. This is necessary for the correctness of any mutual exclusion protocol.

For the purpose of the last requirement one can partition each cycle into a *critical section*, a *noncritical section* (in which the process starts), an *entry protocol* between the noncritical and the critical section, during which a process prepares for entry in negotiation with the competing processes, and an *exit protocol*, that comes right after the critical section and before return to the noncritical section. Now "well outside its critical section" means in the noncritical section. Optionality can equivalently be stated as admitting the possibility that a process chooses to remain forever in its noncritical section, without applying for entry in the critical section ever again.

Knuth [41] proposes a strengthening of the deadlock-freedom requirement, namely

- (*Starvation-freedom*) If a computer intends to enter its critical section, then it will be allowed to enter in due time.

He also presents a solution that is shown to satisfy this requirement, as well as Dijkstra's requirements.[10] Henceforth I define a correct solution of the mutual exclusion problem as one that satisfies both mutex and starvation-freedom, as formulated above, as well as optionality. I speak of "speed-independent mutual exclusion" when also insisting on the requirement of speed independence.

The special case of the mutual exclusion problem for two processes ($N = 2$) was presented by Dijkstra in [15], two years prior to [16]. There Dijkstra presented a solution found by T.J. Dekker in 1959, and shows that it satisfies all requirements of [16]. Although not explicitly stated in [15], the arguments given therein imply straightforwardly that Dekker's solution also satisfies Knuth's starvation-freedom requirement above.

Peterson [52] presented a considerable simplification of Dekker's algorithm that satisfies the same correctness requirements. Many other mutual exclusion protocols appear in the literature, the most prominent being Lamport's bakery algorithm [42] and Szymański's mutual exclusion algorithm [58]. These guarantee some additional correctness criteria besides the ones discussed above.

10. Fair schedulers

In [32] a *fair scheduler* is defined as

> "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

[10] However, Knuth's solution satisfies starvation-freedom, and even deadlock-freedom, only when making a fairness assumption. In fact, all mutual exclusion protocols, including the ones of [15,42,52,58] discussed below, need a fairness assumption to solve the problem as stated by Dijkstra above in a starvation-free way. This will be discussed in Part III of this paper.

FS 1 r_i to accept another request r_i on the same channel. However, the system will always return to a state where it remains ready to accept the next request r_i until r_i 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.

FS 2 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."

FS 3 "We require that in any partial run of the scheduler there may not be more occurrences of t_i than of r_i, for $i = 1, 2$.

FS 4 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."

This fair scheduler serves two clients, but the concept generalises smoothly to N clients.

The intended applications of fair schedulers are for instance in operating systems, where multiple application processes compete for processing on a single core, or radio broadcasting stations, where the station manager needs to schedule multiple parties competing for airtime. In such cases each applicant must get a turn eventually. The event e signals the end of the time slot allocated to an application process on the single core, or to a broadcast on the radio station.

Fair schedulers occur (in suitable variations) in many distributed systems. Examples are *First in First out*,[11] *Round Robin*, and *Fair Queueing* scheduling algorithms[12] as used in network routers [48,49] and operating systems [40], or the *Completely Fair Scheduler*,[13] which is the default scheduler of the Linux kernel since version 2.6.23.

Each action r_i, t_i and e can be seen as a communication between the fair scheduler and one of its clients. In a reactive system such communications will take place only if both the fair scheduler and its client are ready for it. Requirement FS 1 of a fair scheduler quoted above effectively shifts the responsibility for executing r_i to the client. The actions t_i and e, on the other hand, are seen as the responsibility of the fair scheduler. We do not consider the possibility that the fair scheduler fails to execute t_i merely because the client does not collaborate. Hence [32] assumes that the client cannot prevent the actions t_i and e from occurring. It is furthermore assumed that executing the actions r_i, t_i and e takes a finite amount of time only.

A fair scheduler closely resembles a mutual exclusion protocol. However, its goal is not to achieve mutual exclusion. In most applications, mutual exclusion can be taken for granted, as it is physically impossible to allocate the single core to multiple applications at the same time, or the (single frequency) radio sender to multiple simultaneous broadcasts. Instead, its goal is to ensure that no applicant is passed over forever.

It is not hard to obtain a fair scheduler from a mutual exclusion protocol. For suppose we have a mutual exclusion protocol M, serving two processes P_i $(i = 1, 2)$. I instantiate the noncritical section of Process P_i as patiently awaiting the request r_i. As soon as this request arrives, P_i leaves the noncritical section and starts the entry protocol to get access to the critical section. Starvation-freedom guarantees that P_i will reach its critical section. Now the critical section consists of scheduling task t_i, followed by the intermittent activity e. Trivially, the composition of the two processes P_i, in combination with protocol M, constitutes a fair scheduler, in that it meets the above four requirements.

One cannot quite construct a mutual exclusion protocol from a fair scheduler, due to the fact that in a mutual exclusion protocol leaving the critical section is controlled by the client process. For this purpose one would need to adapt the assumption that the client of a fair scheduler cannot block the intermittent activity e into the assumption that the client can postpone this action, but for a finite amount of time only. In this setting one can build a mutual exclusion protocol, serving two processes P_i $(i = 1, 2)$, from a fair scheduler F. Process i simply issues request r_i at F as soon as it has left the noncritical section, and when F communicates the action t_i, Process i enters its critical section. Upon leaving its critical section, which is assumed to happen after a finite amount of time, it participates in the synchronisation e with F. Trivially, this yields a correct mutual exclusion protocol.

11. Formalising the requirements for fair schedulers in reactive LTL

The main reason fair schedulers were defined in [32] was to serve as an example of a realistic class of systems of which no representative can be correctly specified in CCS, or similar process algebras, or in Petri nets. Proving this impossibility result necessitated a precise formalisation of the four requirements quoted in Section 10. Through the provided translations of CCS and Petri nets into LTSs, a fair scheduler rendered in CCS or Petri nets can be seen as a state F in an LTS over the set $\{r_i, t_i, e \mid i = 1, 2\}$ of visible actions; all other actions can be considered internal and renamed into τ.

[11] Also known as First Come First Served (FCFS).
[12] http://en.wikipedia.org/wiki/Scheduling_(computing).
[13] http://en.wikipedia.org/wiki/Completely_Fair_Scheduler.

Let a *partial trace* of a state s in an LTS be the sequence of visible actions encountered on a path starting in s [22]. Now the last two requirements (FS3 and FS4) of a fair scheduler are simple properties that should be satisfied by all partial traces σ of state F:

(FS3) σ contains no more occurrences of t_i than of r_i, for $i = 1, 2$,
(FS4) σ contains an occurrence of e between each two occurrences of t_i and t_j for $i, j \in \{1, 2\}$.

FS4 can be conveniently rendered in LTL:

(FS4) $F \models \mathbf{G}\big(t_i \Rightarrow \mathbf{Y}\big((\neg t_1 \wedge \neg t_2)\mathbf{W}e\big)\big)$

for $i \in \{1, 2\}$. Since FS4 is a safety property, it makes no difference whether and how $\models$ is annotated with B and CC. In [44], Lamport argues against the use of the next-state operator $\mathbf{X}$, as it is incompatible with abstraction from irrelevant details in system descriptions. When following this advice, the weak next-state operator $\mathbf{Y}$ in FS4 can be replaced by $t_i\mathbf{W}$; on Kripke structures distilled from LTSs the meaning is the same.

Unfortunately, FS3 cannot be formulated in LTL, due to the need to keep count of the difference in the number of r_i and t_i actions encountered on a path. However, one could strengthen FS3 into

(FS3′) σ contains an occurrence of r_i between each two occurrences of t_i, and prior to the first occurrence of t_i, for $i \in \{1, 2\}$.

This would restrict the class of acceptable fair schedulers, but keep the most interesting examples. Consequently, the impossibility result from [32] applies to this modified class as well. FS3′ can be rendered in LTL in the same style as FS4:

(FS3′) $F \models \big((\neg t_i)\mathbf{W}r_i\big) \wedge \mathbf{G}\big(t_i \Rightarrow \mathbf{Y}\big((\neg t_i)\mathbf{W}r_i\big)\big)$

for $i \in \{1, 2\}$.

Requirement FS2 involves a quantification over all complete runs of the system, and thus depends on the completeness criterion CC employed. It can be formalised as

(FS2) $F \models_B^{CC} \mathbf{G}(r_i \Rightarrow \mathbf{F}t_i)$

for $i \in \{1, 2\}$, where $B = \{r_1, r_2\}$. The set B should contain r_1 and r_2, as these actions are supposed to be under the control of the users of a fair scheduler. However, actions t_1, t_2 and e should not be in B, as they are under the control of the scheduler itself. In [32], the completeness criterion employed is justness, so the above formula with $CC := J$ captures the requirement on the fair schedulers that are shown in [32] not to exist in CCS or Petri nets. However, keeping CC a variable allows one to pose the question under which completeness criterion a fair scheduler *can* be rendered in CCS. Naturally, it needs to be a stronger criterion than justness. In [32] it is shown that weak fairness suffices.

FS2 is a good example of a requirement that can *not* be rendered correctly in standard LTL. Writing $F \models \mathbf{G}(r_i \Rightarrow \mathbf{F}t_i)$ would rule out the complete runs of F that end because the user of F never supplies the input $r_j \in B$. The CCS process

$$F \stackrel{def}{=} r_1.r_2.t_1.e.t_2.e.F$$

for instance satisfies this formula, due to its unique infinite path, as well as FS3 and 4; yet it does not satisfy Requirement FS2. Namely, the path consisting of the r_1-transition only is complete, since it ends in a state of which the only outgoing transition has the label $r_2 \in B$. It models a (complete) run that can occur when the environment never issues a request r_2, as allowed by FS1. Yet on this path r_1 is not followed by t_1.

Requirement FS1 is by far the hardest to formalise. In [32] two formalisations are shown to be equivalent: one involving a coinductive definition of B-just paths that exploits the syntax of CCS, and the other requiring that Requirements FS2–4 are preserved under putting an input interface around Process F. The latter demands that also

$$\widehat{F} := (I_1 \mid F[f] \mid I_2)\backslash\{c_1, c_2\}$$

should satisfy FS2–4; here f is a relabelling with $f(r_i) = c_i$, $f(t_i) = t_i$ and $f(e) = e$ for $i = 1, 2$, and $I_i \stackrel{def}{=} r_i.\bar{c}_i.I_i$ for $i \in \{1, 2\}$. A similar interface for Petri nets occurs in [39].

A formalisation of FS1 on Petri nets also appears in [32]: each complete path π with only finitely many occurrences of r_i should contain a state (= marking) M, such that there is a transition v with $\ell(v) = r_i$ and ${}^\bullet v \leq M$, and for each transition u that occurs in π past M one has ${}^\bullet v \cap {}^\bullet u = \emptyset$.

When discussing proposals for fair schedulers by others, FS 1 is the requirement that is most often violated, and explaining why is not always easy.

In reactive LTL, this requirement is formalised as

(FS 1) $F \models^{J}_{B\setminus\{r_i\}} \mathbf{GF}r_i$

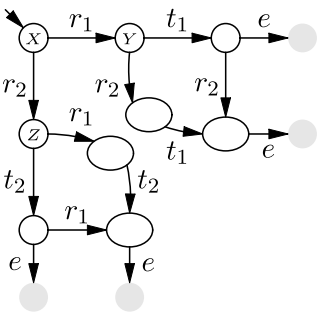
for $i \in \{1,2\}$, or $F \models^{CC}_{B\setminus\{r_i\}} \mathbf{GF}r_i$ if one wants to discuss the completeness criterion CC as a parameter. The surprising element in this temporal judgement is the subscript $B\setminus\{r_i\} = \{r_{3-i}\}$, which contrasts with the assumption that requests are under the control of the environment. FS 1 says that, although we know that there is no guarantee that user i of F will ever issue request r_i, under the assumption that the user *does* want to make such a request, making the request should certainly succeed. This means that the protocol itself does not sit in the way of making this request.

The combination of Requirements FS 1 and 2, which use different sets of blocking actions as a parameter, is enabled by reactive LTL as presented here.

The following examples, taken from [32], show that all the above requirements are necessary for the result from [32] that fair schedulers cannot be rendered in CCS.

- The CCS process $F_1|F_2$ with $F_i \stackrel{def}{=} r_i.t_i.e.F_i$ satisfies FS 1, FS 2 and FS 3′. In FS 1 and 2 one needs to take $CC := J$, as progress is not a strong enough assumption here.
- The process $E_1|G|E_2$ with $E_i \stackrel{def}{=} r_i.E_i$ and $G \stackrel{def}{=} t_1.e.t_2.e.G$ satisfies FS 1, 2 and 4, again with $CC := J$.
- The process $E_1|E_2$ satisfies FS 1, 3′ and 4, again with $CC := J$ in FS 1.
- The process F_0 with $F_0 \stackrel{def}{=} r_1.t_1.e.F_0 + r_2.t_2.e.F_0$ satisfies FS 2–4. Here FS 2 merely needs $CC := Pr$, that is, the assumption of progress. Furthermore, it satisfies FS 1 with $CC := SF(\mathscr{T})$, as long as $\mathrm{R}_1, \mathrm{R}_2 \in \mathscr{T}$. Here R_i is the set of transitions with label r_i.

The process X given by $X \stackrel{def}{=} r_1.Y + r_2.Z$, $Y \stackrel{def}{=} r_2.t_1.e.Z + t_1.(r_2.e.Z + e.X)$ and $Z \stackrel{def}{=} r_1.t_2.e.Y + t_2.(r_1.e.Y + e.X)$, the *gatekeeper*, is depicted on the right. The grey shadows represent copies of the states at the opposite end of the diagram, so the transitions on the far right and bottom loop around. This process satisfies FS 3′ and 4, FS 2 with $CC := Pr$, and FS 1 with $CC := WF(\mathscr{T})$, thereby improving Process F_0, and constituting the best CCS approximation of a fair scheduler seen so far. Yet, intuitively FS 1 is not ensured at all, meaning that weak fairness is too strong an assumption. Nothing really prevents all the choices between r_2 and any other action a to be made in favour of a.



12. Formalising requirements for mutual exclusion in reactive LTL

Define a process i participating in a mutual exclusion protocol to cycle through the stages *noncritical section*, *entry protocol*, *critical section*, and *exit protocol*, in that order, as explained in Section 9. Modelled as an LTS, its visible actions will be en_i, ln_i, ec_i and lc_i, of entering and leaving its (non)critical section. Put ln_i in B to make leaving the noncritical section a blocking action. The environment blocking it is my way of allowing the client process to stay in its noncritical section forever. This is the manner in which the requirement *optionality* is captured in reactive temporal logic. On the other hand, ec_i should not be in B, for one does not consider the starvation-freedom property of a mutual exclusion protocol to be violated simply because the client process refuses to enter the critical section when allowed by the protocol. Likewise, en_i is not in B. Although exiting the critical section is in fact under control of the client process, it is assumed that it will not stay in the critical section forever. In the models of this paper this can be simply achieved by leaving lc_i outside B. Hence $B := \{ln_i \mid i = 1, \ldots, N\}$.

My first requirement on mutual exclusion protocols P simply says that the actions en_i, ln_i, ec_i and lc_i have to occur in the right order:

(ORD) $P \models \big((\neg act_i)\mathbf{W}ln_i\big) \wedge \mathbf{G}\big(ln_i \Rightarrow \mathbf{Y}\big((\neg act_i)\mathbf{W}ec_i\big)\big) \wedge \mathbf{G}\big(ec_i \Rightarrow \mathbf{Y}\big((\neg act_i)\mathbf{W}lc_i\big)\big)$
$\wedge\, \mathbf{G}\big(lc_i \Rightarrow \mathbf{Y}\big((\neg act_i)\mathbf{W}en_i\big)\big) \wedge \mathbf{G}\big(en_i \Rightarrow \mathbf{Y}\big((\neg act_i)\mathbf{W}ln_i\big)\big)$

for $i = 1, \ldots, N$. Here $act_i := (ln_i \vee ec_i \vee lc_i \vee en_i)$.

The second is a formalisation of *mutex*, saying that only one process can be in its critical section at the same time:

(ME) $P \models \mathbf{G}\big(ec_i \Rightarrow \big((\neg ec_j)\mathbf{W}lc_i\big)\big)$

for all $i, j = 1, \ldots, N$ with $i \neq j$. Both ORD and ME are safety properties, and thus unaffected by changing $\models$ into $\models^{CC}$ or $\models^{CC}_{B}$.

The starvation-freedom requirement of Section 9 can be formalised as

(EC^{CC}) $P \models_B^{CC} \mathbf{G}(ln_i \Rightarrow \mathbf{F}ec_i)$

for $i = 1, \ldots, N$. Here the choice of a completeness criterion is important. Finally, the following requirements are similar to starvation-freedom, and state that from each section in the cycle of a process i, the next section will in fact be reached. In regards to reaching the end of the noncritical section, this should be guaranteed only when assuming that the process wants to leave its noncritical section; hence ln_i is excepted from B.

(LC^{CC}) $P \models_B^{CC} \mathbf{G}(ec_i \Rightarrow \mathbf{F}lc_i)$
(EN^{CC}) $P \models_B^{CC} \mathbf{G}(lc_i \Rightarrow \mathbf{F}en_i)$
(LN^{CC}) $P \models_{B\setminus\{ln_i\}}^{CC} \mathbf{F}ln_i \wedge \mathbf{G}(en_i \Rightarrow \mathbf{F}ln_i)$

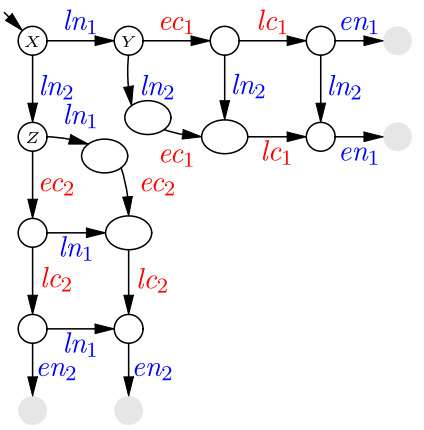
for $i = 1, \ldots, N$.

The requirement *speed independence* is automatically satisfied for models of mutual exclusion protocols rendered in any of the formalisms discussed so far, as these formalisms lack the expressiveness to make anything dependent on speed.

The following examples show that none of the above requirements are redundant.

- The CCS process $F_1|F_2|\cdots|F_N$ with $F_i \stackrel{def}{=} ln_i.ec_i.lc_i.en_i.F_i$ satisfies all requirements, with $CC := J$, except for ME.
- The process $R_1|R_2|\cdots|R_N$ with $R_i \stackrel{def}{=} ln_i.\mathbf{0}$ satisfies all requirements except for EC.
- For $N = 2$, process $H \stackrel{def}{=} ln_1.ec_1.ln_2.lc_1.en_1.ec_2.lc_2.en_2.H + ln_2.ec_2.ln_1.lc_2.en_2.ec_1.lc_1.en_1.H$ satisfies all requirements but LC. The case $N > 2$ is only notationally more cumbersome. Similarly, one finds examples failing only on EN, or on the second conjunct of LN.
- The process $\mathbf{0}$ satisfies all requirements except for the first conjunct of LN.
- In case $N = 1$, the process $W \stackrel{def}{=} lc_1.ec_1.lc_1.en_1.ln_1.W$ satisfies all requirements but ORD.

The process X, a gatekeeper variant, given by $X \stackrel{def}{=} ln_1.Y + ln_2.Z$, $Y \stackrel{def}{=} ln_2.ec_1.lc_1.en_1.Z + ec_1.(ln_2.lc_1.en_1.Z + lc_1.(ln_2.en_1.Z + en_1.X))$ $Z \stackrel{def}{=} ln_1.ec_2.lc_2.en_2.Y + ec_2.(ln_1.lc_2.en_2.Y + lc_2.(ln_1.en_2.Y + en_2.X))$ is depicted on the right. It satisfies ORD, ME, EC^{CC}, LC^{CC} and EN^{CC} with $CC := Pr$ and LN with $CC := WF(\mathscr{T})$, where $\mathrm{LN}_1, \mathrm{LN}_2 \in \mathscr{T}$. It can be seen as a mediator that synchronises, on the actions ln_i, ec_i, lc_i and en_i, with the actual processes that need to exclusively enter their critical sections. Yet, it would not be commonly accepted as a valid mutual exclusion protocol, since nothing prevents it to never choose ln_2. This means that merely requiring weak fairness in LN makes this requirement unacceptably weak. The problem with this protocol is that it ensures starvation-freedom by making it hard for processes to leave their noncritical sections.



13. State-oriented requirements for mutual exclusion

Some readers may prefer a state-oriented view of mutual exclusion over the action-oriented view of Section 12. In such a view, the mutex requirement (ME in Section 12) simply says that different processes i and j cannot *be* in the critical section at the same time, rather than encoding this in terms of the actions of entering and leaving the critical section. This section translates the requirements on mutual exclusion from the action-oriented view of Section 12 to a state-oriented view.

Let's model the protocol with a Kripke structure that features the atomic predicates C_i and I_i, for $i = 1, 2$. Predicate C_i holds when Process i is in its critical section, and I_i when Process i intends to enter its critical section, but isn't there yet. Predicate I_i should thus hold when Process i has left the noncritical section and is executing its entry protocol. In this presentation one can lump together the exit protocol and the noncritical section of process i; these comprise the states satisfying $\neg(I_i \vee C_i)$. Now Requirements ORD–LN can be reformulated as follows, for $i, j = 1, \ldots, N$.

(ORD) $P \models \mathbf{G}\neg(C_i \wedge I_i) \wedge \neg(I_i \vee C_i) \wedge \mathbf{G}(I_i \Rightarrow (I_i \mathbf{W} C_i)) \wedge \mathbf{G}(C_i \Rightarrow (C_i \mathbf{W} \neg(I_i \vee C_i)))$
$\wedge \mathbf{G}(\neg(I_i \vee C_i) \Rightarrow (\neg(I_i \vee C_i) \mathbf{W} I_i))$
(ME) $P \models \mathbf{G}\big(\neg(C_i \wedge C_j)\big)$ for all $j \neq i$
(EC^{CC}) $P \models_B^{CC} \mathbf{G}(I_i \Rightarrow \mathbf{F}C_i)$
(LC^{CC}) $P \models_B^{CC} \mathbf{G}(C_i \Rightarrow \mathbf{F}\neg(I_i \vee C_i))$
(LN^{CC}) $P \models_{B\setminus\{I_i\}}^{CC} \mathbf{G}(\neg(I_i \vee C_i) \Rightarrow \mathbf{F}I_i)$

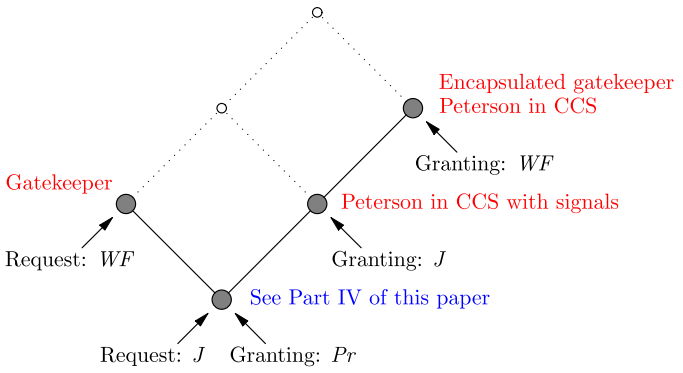
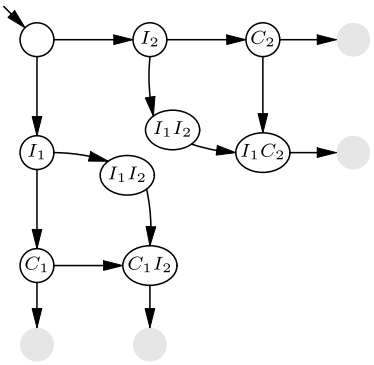


Fig. 3. A hierarchy of quality criteria for fair schedulers and mutual exclusion protocols.

Here B can be rendered as a set of atomic predicates p, and refers to all those "blocking transitions" that go from a state where p does not hold to one where p holds, for some $p \in B$. So here $B = \{I_i \mid i = 1, \ldots, N\}$.

In this setting, the gatekeeper is depicted on the right. It satisfies ORD, ME, EC^{CC} and LC^{CC} above, with $CC = Pr$, but LN^{CC} only with $CC = WF$.



14. A hierarchy of quality criteria for mutual exclusion protocols

Formalising the quality criteria for fair schedulers as FS1–4, one sees that, unlike FS3 and 4, Requirements FS1 and 2 are parametrised by the choice of a completeness criterion CC. In each of FS1 and FS2, CC can be instantiated with either $\top$, Pr, J, $WF(\mathscr{T})$ or $SF(\mathscr{T})$ for a suitable collection of tasks $\mathscr{T}$. When seeing $WF(\mathscr{T})$ or $SF(\mathscr{T})$ as single choices, allowing them to utilise the most appropriate choice of $\mathscr{T}$, this yields a hierarchy of $5 \times 5 = 25$ different quality criteria for fair schedulers, partially depicted in Fig. 3. Here "Request" indicates the completeness criterion used in Requirement FS1 and "Granting" the one taken in FS2. Note that quality criteria encountered further up in the figure employ stronger fairness assumptions, and thus yield weaker, or less impressive, fair schedulers.

I have not rendered all 25 quality criteria in Fig. 3, as many are irrelevant. Since no meaningful liveness property holds when merely assuming the trivial completeness criterion $\top$, one can safely discard it from consideration; there can exist no fair scheduler satisfying FS1 or 2 with $CC = \top$. Likewise, one can forget about the possibility "Request: Pr". As an infinite run such as $(r_2t_2e)^\infty$ in which a request r_1 is never received, and consequently a request-granting action t_1 never occurs, should be a complete run of the system, progress is not strong enough an assumption to ensure that when user 1 wants to issue request r_1 it will actually succeed. The least one should assume here is justness.

At the other end of the hierarchy I have dropped the choice SF. The reason is that there turn out to be completely satisfactory solutions merely assuming weak fairness in either dimension; so the choice of strong fairness makes the fair scheduler unnecessary weak.

The same hierarchy of quality criteria applies to mutual exclusion protocols. Now "Request" indicates the completeness criterion used in Requirement LN and "Granting" the one taken in EC. The latter concerns the starvation-freedom property of mutual exclusion; it indicates how hard it is to reach the critical section after a process' interest in doing this has been expressed. The former indicates how hard it is to express such an interest in the first place. Again, the choice "Request: Pr" can be discarded, as no mutual exclusion protocol can meet this requirement. This is due to the infinite run $(ln_2ec_2lc_2en_2)^\infty$, in which process 1 never requests access to the critical section. When merely assuming progress, one cannot tell whether this is because process 1 does not want to leave its noncritical section, or because it wants to but doesn't succeed, as always another action is chosen.

In principle, there are two more dimensions in classifying the quality criteria for mutual exclusion protocols, namely the choice of a completeness criterion for Requirements LC and EN. These indicate how hard it is to leave the critical section after entering, and to enter the noncritical section after leaving the critical one, respectively. As both tasks are really easy to accomplish, these two dimensions are not indicated in Fig. 3. Nevertheless, they should not be forgotten in the forthcoming analysis. There are no further dimensions for ORD and ME, as these are safety properties.

When allowing weak fairness in the "request" dimension, the gatekeeper, described for fair schedulers in Section 11 and for mutual exclusion in Section 12, is a good solution. It merely requires progress in the "granting" dimension, and for mutual exclusion also in LC and EN. As most researchers in the area of mutual exclusion would agree that nevertheless the gatekeeper is not an acceptable protocol, we have evidence that weak fairness in the "request" dimension is too strong an assumption.

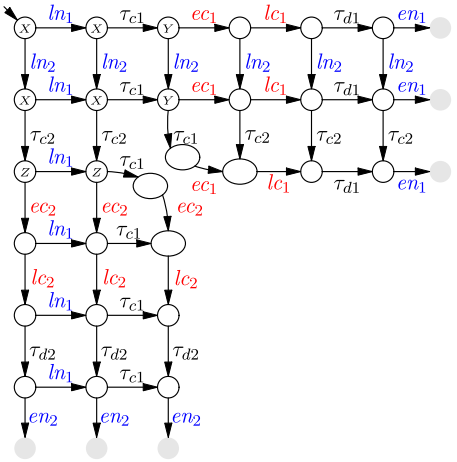


Fig. 4. Encapsulated gatekeeper.

15. An input interface for implementing LN

In [32, Section 13] an input interface is proposed that can be put around any potential fair scheduler expressed in CCS—it was recalled in Section 11. It was shown that a process F satisfies Requirements FS1–4 iff the *encapsulated* process $\widehat{F}$—the result of putting F in this interface—satisfies FS2–4. Here I propose a similar interface for mutual exclusion protocols, restricting attention to the case of $N = 2$ parties.

Definition 15.1. For any expression P, let $\widehat{P} := (I_1 \mid P[f] \mid I_2)\backslash\{c_1, c_2, d_1, d_2\}$ where $I_i \stackrel{def}{=} ln_i.\bar{c}_i.\bar{d}_i.en_i.I_i$ for $i \in \{1, 2\}$ and f is a relabelling with $f(ln_i) = c_i$ and $f(en_i) = d_i$ for $i \in \{1, 2\}$, such that $f(a) = a$ for all other actions a occurring in P.

Observation 15.2. Suppose that P satisfies ORD, and criterion CC is at least as strong as justness. Then $\widehat{P}$ satisfies ORD as well as LN^{J}, and

- $\widehat{P}$ satisfies ME iff P satisfies ME.
- $\widehat{P}$ satisfies EC^{CC} iff P satisfies LN^{CC} and EC^{CC},
- $\widehat{P}$ satisfies LC^{CC} iff P satisfies LC^{CC}.
- $\widehat{P}$ satisfies EN^{CC} iff P satisfies EN^{CC}.

Let the *encapsulated gatekeeper* H be the result of putting this input interface around the gatekeeper for mutual exclusion. It can be described as follows, and its labelled transition system is depicted in Fig. 4. Here I added subscripts to τ-action to indicate their origins.

$$
\begin{aligned}
&H := (I_1 \mid X \mid I_2)\backslash\{c_1, c_2, d_1, d_2\}\\
&I_i \stackrel{def}{=} ln_i.\bar{c}_i.d_i.en_i.I_i \qquad \text{for } i \in \{1, 2\},\\
&X \stackrel{def}{=} c_1.Y + c_2.Z,\\
&Y \stackrel{def}{=} c_2.ec_1.lc_1.\bar{d}_1.Z + ec_1.(c_2.lc_1.\bar{d}_1.Z + lc_1.(c_2.\bar{d}_1.Z + \bar{d}_1.X))\\
&Z \stackrel{def}{=} c_1.ec_2.lc_2.\bar{d}_2.Y + ec_2.(c_1.lc_2.\bar{d}_2.Y + lc_2.(c_1.\bar{d}_2.Y + \bar{d}_2.X))
\end{aligned}
$$

This mutual exclusion protocol satisfies LN^{J}, for as soon as en_i has occurred (and in the initial state) Process I_i is in its initial state $ln_i.\bar{c}_i.d_i.en_i.I_i$, and nothing stands in the way of the action ln_i. In other words, justness is a strong enough assumption for ln_i to occur. Clearly, the protocol also satisfies ORD and ME, as well as LC^{Pr} and LC^{Pr}. The only downside is that it takes weak fairness to achieve EC, starvation-freedom. This assumption is needed to assure that the synchronisation between actions $\bar{c}_i$ and c_i will actually occur.

Intuitively, the encapsulated gatekeeper is an equally unacceptable mutual exclusion protocol as the gatekeeper, for the input interface ought to make no difference. This shows that weak fairness in any dimension of Fig. 3 is too strong an assumption. However, due to the impossibility result of [32], the two remaining entries of Fig. 3 cannot be realised in CCS. Theoretically, that result leaves open the possibility of achieving justness in both the dimensions "request" and "granting", at the expense of assuming weak fairness for LC or EN. I do not think this is actually possible, and even if it were, a solution that requires weak fairness to escape the critical section, or to enter the noncritical one, appears equally unacceptable as the (encapsulated) gatekeeper.

Part III. Impossibility results for Peterson's mutual exclusion algorithm

Here I recall three impossibility results for mutual exclusion protocols that have been shown or claimed earlier, and illustrate or substantiate them for Peterson's mutual exclusion protocol. I could have equally well done this for another mutual exclusion protocol, such as Lamport's bakery algorithm [42], Aravind's mutual exclusion algorithm [2] or the *round-robin* scheduler.[14] My reason for choosing Peterson's protocol in this paper is that it is (one of) the simplest of all mutual exclusion protocols.

The first impossibility result stems from [60,39,32], and says that in Petri nets, and in CCS and similar process algebras, it is not possible to model a mutual exclusion protocol in such a way that it is correct without making an assumption as strong as weak fairness. In [60] this is shown for finite Petri nets, and in [39] for a class of Petri nets that interact with their environment through an interface of a particular shape, similar to the one of Section 15. In [32] the same is shown for all structural conflict nets (and thus for all safe nets), as well as for the process algebra CCS, with strong hints on how the result extends to many similar process algebras. For the latter result, either the concurrency relation between CCS transitions defined in Section 4.4, or directly the resulting concept of a just path, needs to be seen as an integral part of CCS. In Sections 18 and 19 I will illustrate this impossibility result for a rendering of Peterson's protocol as a Petri net and as a CCS expression, respectively. In [32,33] moreover the point of view is defended that assuming (strong or weak) fairness is typically unwarranted, in the sense that there is no reason to assume that reality will behave in a fair way. From this point of view, a model of a mutual exclusion that hinges on a fairness assumption can be seen as incorrect or unsatisfactory. This makes the above into a real impossibility result.

The second impossibility result was claimed in [24], but unaccompanied by written evidence. It blames the first impossibility result above on the combination of two assumptions or protocol features, which I here call *atomicity* and *speed independence*. Atomicity, or rather the special case of atomicity that is relevant for the second impossibility result, will be formally defined as (1) in Section 20. It can be seen as an assumption on the behaviour of the hardware on which an implementation of a mutual exclusion protocol will be running. Atomicity is explicitly assumed in the original paper of Dijkstra where the mutual exclusion problem was presented [16], and implicitly in many other papers on mutual exclusion, but not in the work of Lamport [42]. Speed independence can either be seen as an assumption on the underlying hardware, or as a feature of a mutual exclusion protocol. The assumption stems from Dijkstra [16] and was quoted in Section 9. The claims of [24] employ a rather strict interpretation of speed independence, illustrated in Example 22.1.

In a setting where solutions based on the assumption of weak fairness are rejected, as well as solutions that are merely probabilistically correct, [24] claims that when assuming atomicity, speed-independent mutual exclusion is impossible. This means that when assuming atomicity and speed independence, there is no mutual exclusion protocol satisfying ORD–LN with $CC = J$. The assumption of speed independence is built in in CCS and Petri nets, in the sense that any correct mutual exclusion protocol formalised therein is automatically speed independent. This is because these models lack the expressiveness to make anything dependent on speed. When taking the concurrency relation between CCS or net transitions defined in Section 4.4 as an integral part of semantics of CCS or Petri nets, also the assumption of atomicity is built in in these frameworks. This makes the first impossibility result a special case of the second. The latter can be seen as a generalisation of the former that is not dependent on a particular modelling framework. In Section 22 I will substantiate the above claim of [24] for the special case of Peterson's protocol.

The third impossibility result, also claimed in [24], says that when dropping the assumption of atomicity, but keeping speed independence, there still exists no mutual exclusion protocol satisfying EC^{Pr}. That is, the assumption of progress is not strong enough to obtain starvation-freedom of any speed-independent mutual exclusion protocol. I will substantiate this claim for the special case of Peterson's protocol in Section 16. In Part IV I aim for a formalisation of Peterson's protocol that satisfies EC^{Pr}. It follows that there I will have to drop speed independence.

16. Peterson's mutual exclusion protocol

A pseudocode rendering of Peterson's protocol is depicted in Fig. 5. The two processes, here called A and B, use three shared variables: *readyA*, *readyB* and *turn*. The Boolean variable *readyA* can be written by Process A and read by Process B, whereas *readyB* can be written by B and read by A. By setting *readyA* to *true*, Process A signals to Process B that it wants to enter the critical section. The variable *turn* can be written and read by both processes. Its carefully designed functionality guarantees mutual exclusion as well as deadlock-freedom. Both *readyA* and *readyB* are initialised with *false* and *turn* with *A*.

Fig. 6 presents a labelled transition system for this protocol. The name of a state $\ell_i{}^{T}_{m_j}$ where T is A or B indicates that Process A is in State ℓ_i, Process B in State m_j and the variable *turn* has value T, with the convention that Instruction ℓ_i leads from State ℓ_i to State ℓ_{i+1}. This completely determines the values of the variables *readyA* and *readyB*. The actions ln_A, ln_B,

[14] As a mutual exclusion protocol, the round-robin is a central scheduler that grants access to the critical section to N processes numbered $1-N$ by cycling through all competing processes in the order $1-N$. Each time it is the turn of Process i, the round-robin scheduler checks whether Process i wants to enter the critical section, and if so, grants access. When Process i leaves the critical section, or if it didn't want to enter, it will be the turn of Process $i{+}1 \bmod N$.

When confronted with the claim that under some natural assumptions no correct mutual exclusion protocols exist, some people reply that in that case one could always use a round-robin scheduler, as if this somehow constitutes an exception.

Process A

repeat forever

ℓ_1 **leave noncritical section**
ℓ_2 *readyA* := *true*
ℓ_3 *turn* := *B*
ℓ_4 **await** (*readyB* = *false* ∨ *turn* = *A*)
ℓ_5 **enter critical section**
ℓ_6 **leave critical section**
ℓ_7 *readyA* := *false*
ℓ_8 **enter noncritical section**

Process B

repeat forever

m_1 **leave noncritical section**
m_2 *readyB* := *true*
m_3 *turn* := *A*
m_4 **await** (*readyA* = *false* ∨ *turn* = *B*)
m_5 **enter critical section**
m_6 **leave critical section**
m_7 *readyB* := *false*
m_8 **enter noncritical section**

Fig. 5. Peterson's algorithm (pseudocode).

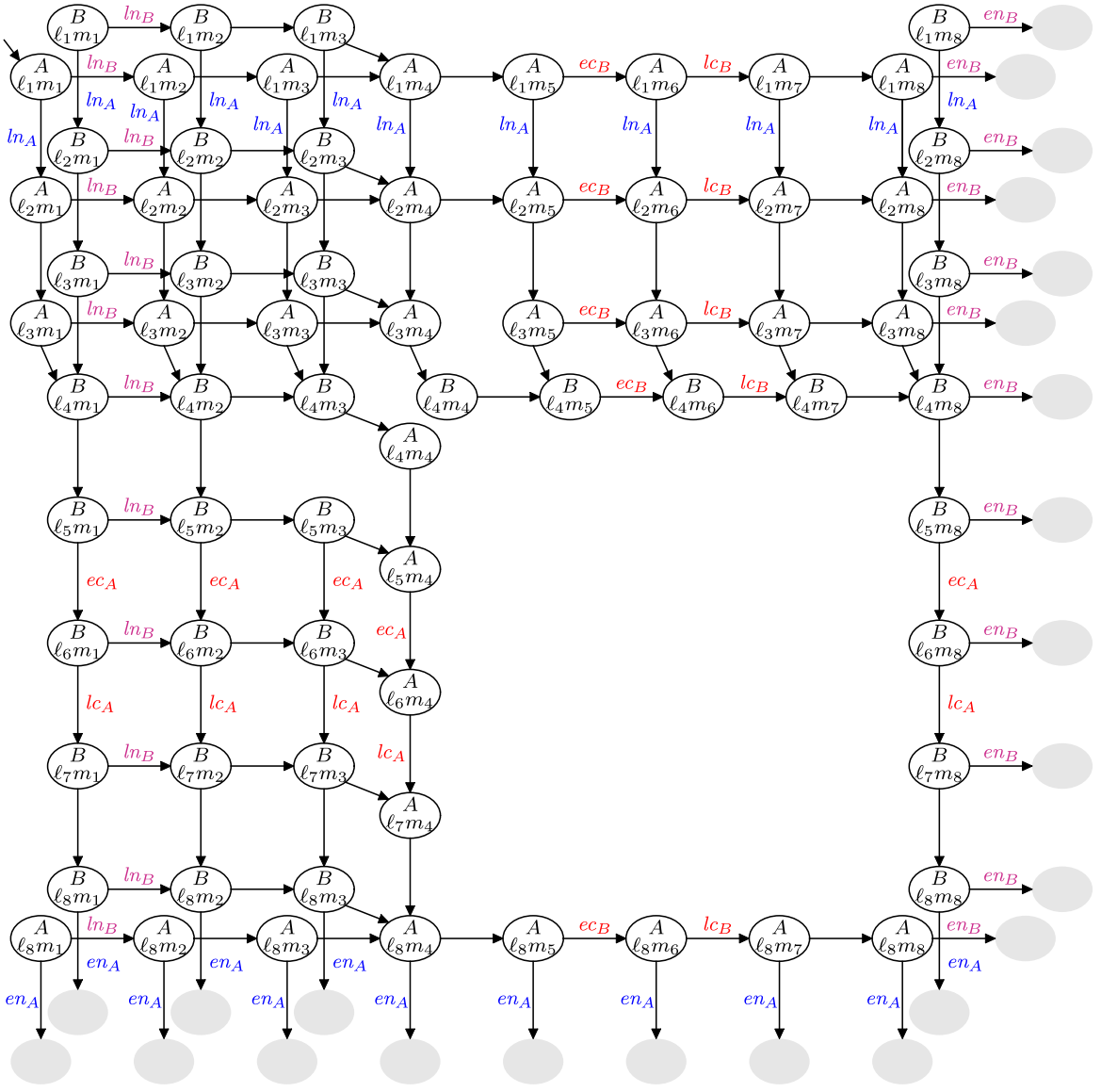


Fig. 6. LTS of Peterson's mutual exclusion algorithm.

ec_A, ec_B, lc_A, lc_B, en_A and en_B are visible; they correspond to the instructions ℓ_1, m_1, ℓ_5, m_5, ℓ_6, m_6, ℓ_8 and m_8, respectively. All other transitions, unlabelled in Fig. 6, are labelled τ. When assuming speed independence, none of the paths in Fig. 6 can be ruled out due to timing considerations. This makes Fig. 6 an adequate rendering of Peterson's protocol.

From the pseudocode in Fig. 5 one sees immediately that Peterson's protocol satisfies ORD (all visible actions occur in the right order) and LN^J (nothing stands in the way of a process leaving its noncritical section). By inspecting the LTS in Fig. 6 one sees that it moreover satisfies ME (the mutual exclusion property) and LC^{Pr} (assuming progress and willingness to do so is enough to ensure that a process will leave its critical section after entering). One obtains EN^J (assuming justness suffices to ensure that a process always enters its noncritical section after leaving its critical section) by combining the code and the LTS. The LTS shows that assuming progress suffices to ensure that lc_B is always followed by m_7. The code shows that assuming justness suffices to ensure that m_7 is always followed by ln_B. Of course the same applies to Process A.

More problematic is Requirement EC, starvation-freedom. Thanks to symmetry, I may restrict attention to EC for Process A. Will Instruction $\ell_1 = ln_A$ always be followed by $\ell_5 = ec_A$? The LTS shows that ℓ_2 is always followed by $\ell_5 = ec_A$, even when merely assuming progress. However, it is less clear whether $\ell_1 = ln_A$ is always followed by ℓ_2. The only[15] progressing path π_P on which ℓ_1 is not followed by ℓ_2 visits state $\overset{A}{\ell_2 m_4}$ infinitely often and always takes the transition going right. This path witnesses that progress is not strong enough an assumption to ensure EC, whereas weak fairness is, provided all the transitions stemming from Instruction ℓ_2 form a task.[16] Whether justness is a strong enough assumption for EC depends solely on the question whether π_P is just.

To answer that question one can interpret the LTS of Fig. 6 as an LTSC, by investigating an appropriate concurrency relation $\frown$ on the transitions. Whether two transitions are concurrent ought to depend solely on the instructions ℓ_i and m_j from Fig. 5 that gave rise to these transitions, that is, whether $\ell_i \frown m_j$ for $i, j = 1, \ldots, 8$. Assuming that the three variables *readyA*, *readyB* and *turn* are stored in independent stores or registers, the only pairs that may violate $\ell_i \frown m_j$ are $\ell_3 \not\frown m_3$, $\ell_2 \not\frown m_4$, $\ell_3 \not\frown m_4$, $\ell_7 \not\frown m_4$, $\ell_4 \not\frown m_2$, $\ell_4 \not\frown m_3$ and $\ell_4 \not\frown m_7$, for these instruction compete for access to the same register.[17] The only pair out of these 7 that affects the justness of π_P is $\ell_2 \not\frown m_4$. Considering that in State $\overset{A}{\ell_2 m_4}$ it is possible to perform Instruction m_4, moving to State $\overset{A}{\ell_2 m_5}$, yet in State $\overset{A}{\ell_3 m_4}$, thus after performing ℓ_2, it is no longer possible to perform Instruction m_4, one surely has $m_4 \not\frown \ell_2$, that is, Instruction m_4 is affected by ℓ_2—compare (4.2). On the other hand, one cannot derive from Fig. 6 alone whether $\ell_2 \not\frown m_4$. In case one decides that $\ell_2 \frown m_4$ then π_P is not B-just by Definition 6.1; as nothing blocks the execution of Instruction ℓ_2, it must eventually occur. In this case EC^J holds. However, in case one decides that $\ell_2 \not\frown m_4$, then π_P is just and EC^J does not hold.

Sections 18–19 examine whether $\ell_2 \frown m_4$ holds in renderings of this protocol as a Petri net and in CCS. In Section 20 I will reflect on whether $\ell_2 \frown m_4$ holds, and thus on whether Peterson's protocol satisfies EC^J, based on a classification as to what could happen if two processes try to access the same register at the same time, one writing and one reading.

17. Verifications of starvation-freedom merely assuming progress

Above, I argued that in any formalisation P of Peterson's algorithm that is consistent with the LTS of Fig. 6—including any speed-independent formalisation—one has $P \not\models_B^{Pr} \mathbf{G}(ln_i \Rightarrow \mathbf{F} ec_i)$, that is, Requirement EC^{Pr} does not hold, or, the formalisation P does not satisfy starvation-freedom when merely assuming progress. The path π_P from Section 16 constitutes a counterexample. In Part IV of this paper I will bypass this verdict by proposing a formalisation of Peterson's algorithm that is not consistent with the LTS of Fig. 6.

In [61], Peterson's algorithm was formalised in a way that is consistent with Fig. 6, yet starvation-freedom was proven, even by automatic means, while assuming no more than progress. The contradiction with the above is only apparent, because the starvation-freedom property obtained by [61] can be stated as $P \not\models_B^{Pr} \mathbf{G}(\ell_2 \Rightarrow \mathbf{F} ec_A)$, and symmetrically $P \not\models_B^{Pr} \mathbf{G}(m_2 \Rightarrow \mathbf{F} ec_B)$, that is, once a process expresses its intention to enter its critical section, by executing Instruction ℓ_2 (or m_2), then it will surely reach its critical section.[18] The same can be said for the verification of starvation-freedom of Peterson's algorithm in [59], and, in essence, for the verification of starvation-freedom of Dekker's algorithm in [18].

It can be (and has been) debated which is the better formalisation of starvation-freedom. Since the greatest hurdle in the protocol is Instruction ℓ_2, it seems unfair to say that a process is only deemed interested in entering the critical section when this hurdle is taken.[19] Another tactic is to consider a form of Peterson's protocol in which Processes A and B are merely interfaces that interact with the real clients that compete for access to the critical section by synchronising on the

[15] when considering two paths essentially the same if they differ merely on a finite prefix.

[16] Formally, $Peterson \models_B^{WF(\mathscr{T})} \mathbf{G}(ln_i \Rightarrow \mathbf{F} ec_i)$ when $\mathrm{L}_2, \mathrm{M}_2 \in \mathscr{T}$, where L_2 (resp. M_2) contains all transitions stemming from instruction ℓ_2 (resp. m_2). Namely, π_P fails to be weakly fair, for it has a suffix on which task L_2 is perpetually enabled but never occurs.

[17] The verdicts $\ell_1 \frown m_j$, $\ell_8 \frown m_j$, $\ell_i \frown m_1$ and $\ell_i \frown m_8$ were already used implicitly in the above derivation of LN^J and EN^J from the pseudocode.

[18] In the literature [55,54] it is frequently claimed that once Process A expresses its intention to enter its critical section, by executing Instruction ℓ_2, Process B will enter the critical section at most once before A does (and the same with A and B reversed, of course). A counterexample is provided by the path that turns right as much as possible from the state $\overset{A}{\ell_3 m_5}$. Here Process A has just executed ℓ_2, but Process B enters the critical section twice before A does. (By the definitions in [54], in state $\overset{A}{\ell_3 m_5}$ the two processes are *competing*, and A loses the competition twice.)

[19] Suppose I promise all of you $1000, if only you express interest in getting it, by filling in Form 316F. Then I implement this promise by making it impossible to fill in Form 316F. In that case you might argue against the claim that the Requirement $\mathbf{G}$("has interest" $\Rightarrow \mathbf{F}$"receive $1000") can be verified. The argument would be that filling in Form 316F is an inadequate formalisation of having interest in getting this prize.

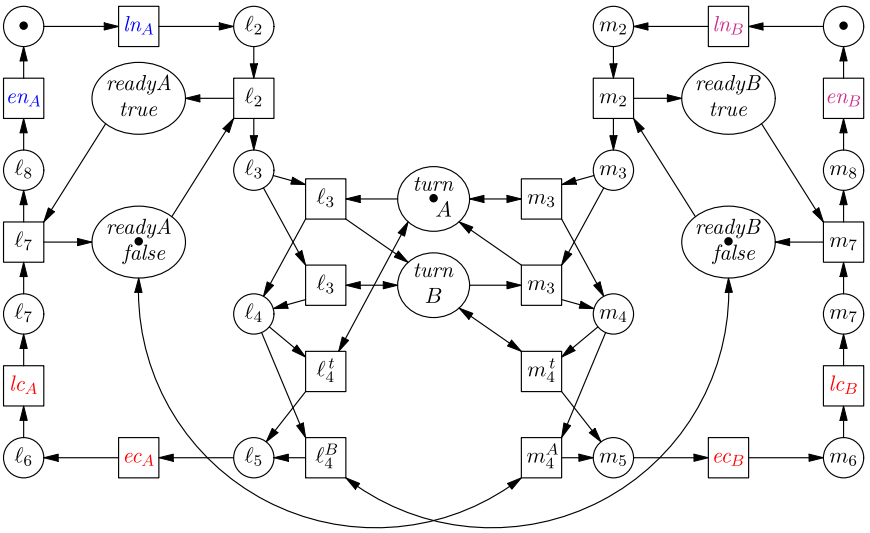


Fig. 7. Petri net representation of Peterson's mutual exclusion algorithm.

actions ln_A, ln_B, ec_A, ec_B, lc_A, lc_B, en_A and en_B. In such a case, the real intent of Client A to enter its critical section is expressed in a message to Interface A that occurs strictly before Interface A executes Instruction ℓ_2.

Such arguments are less necessary now that I have formalised LN as an additional requirement. Suppose that one would redefine the action ln_A that appears in the antecedent of the starvation-freedom requirement EC as the occurrence of instruction ℓ_2 from Peterson's algorithm, thus turning EC for Process A into $P \models_B^{CC} \mathbf{G}(\ell_2 \Rightarrow \mathbf{F}ec_A)$, then LN becomes $P \models_{B\setminus\{\ell_2\}}^{CC} \mathbf{F}\ell_2 \wedge \mathbf{G}(en_A \Rightarrow \mathbf{F}\ell_2)$. Either way, it is required that ℓ_1 will be followed by ℓ_2; this requirement is either part of LN or of EC. In case it takes weak fairness to perform Instruction ℓ_2, either EC or LN will hold only for $CC = WF$, and depending on one's modelling preferences one can choose which one.

In Sections 18 and 19 I will show that in formalisations of Peterson's algorithm as a Petri net or a CCS expression, the path π_P from Section 16 is just, implying that it takes weak fairness to perform Instruction ℓ_2. This implies that those formulations can in terms of Fig. 3 be situated either at the coordinates (WF, Pr) or (J, WF), depending on one's preferred modelling of intent to enter the critical section. Either way, such a rendering of Peterson scores no better than the (encapsulated) gatekeeper.

18. Modelling Peterson's protocol as a Petri net

Fig. 7 shows a rendering of Peterson's protocol as a Petri net. There is one place for each of the local states ℓ_1–ℓ_8 and m_1–m_8 of Processes A and B and two for each of the Boolean variables *readyA*, *readyB* and *turn*. There is one transition for each of the instructions ℓ_1–ℓ_8 and m_1–m_8, except that ℓ_3, m_3, ℓ_4 and m_4 yield two transitions each. For ℓ_3 and m_3 this is to deal with each of the possible values of *turn* before the assignment is executed; for ℓ_4 the two transitions are for reading that $turn = A$, and for reading that $readyB = false$.

The transitions ℓ_2 and m_4^A are not concurrent, because they compete for the same token on the place $readyA = false$. For this reason the run π_P, in which one token is stuck in place ℓ_2 while the other four tokens keep moving around—with m_4^A executed infinitely many times—is just. Consequently, this Petri net does not satisfy requirement EC^J.

19. Modelling Peterson's protocol in CCS

In order to model Peterson's mutual exclusion protocol in CCS, I use the names en_A, en_B, ln_A, ln_B, ec_A, ec_B, lc_A and lc_B for Processes A and B entering and leaving their (non)critical section. Following [17], I describe a simple shared memory system in CCS, using the name $asgn_x^v$ for the assignment of value v to the variable x, and n_x^v for noticing or notifying that the variable x has the value v. The action $\overline{asgn_x^v}$ communicates the assignment $x := v$ to the shared memory, whereas $asgn_x^v$ is the action of the shared memory of accepting this communication. Likewise, $\overline{n_x^v}$ is a notification by the shared memory that x equals v; it synchronises with the complementary action n_x^v of noticing that $x = v$.

The Processes A and B can be modelled as

$$\mathrm{A} \stackrel{def}{=} ln_A \,.\, \overline{asgn^{true}_{readyA}} \,.\, \overline{asgn^{B}_{turn}} \,.\, (n^{false}_{readyB} + n^{A}_{turn}) \,.\, ec_A \,.\, lc_A \,.\, \overline{asgn^{false}_{readyA}} \,.\, en_A \,.\, \mathrm{A}\,,$$
$$\mathrm{B} \stackrel{def}{=} ln_B \,.\, \overline{asgn^{true}_{readyB}} \,.\, \overline{asgn^{A}_{turn}} \,.\, (n^{false}_{readyA} + n^{B}_{turn}) \,.\, ec_B \,.\, lc_B \,.\, \overline{asgn^{false}_{readyB}} \,.\, en_B \,.\, \mathrm{B}\,,$$

where $(a+b).P$ is a shorthand for $a.P + b.P$. This CCS rendering naturally captures the **await** statement, requiring Process A to wait at Instruction ℓ_4 until it can read that $readyB = false$ or $turn = A$. We use two agent identifiers for each Boolean variable x, one for each value:

$$x^{true} \stackrel{def}{=} asgn_x^{true}.x^{true} + asgn_x^{false}.x^{false} + \overline{n_x^{true}}.x^{true},$$
$$x^{false} \stackrel{def}{=} asgn_x^{true}.x^{true} + asgn_x^{false}.x^{false} + \overline{n_x^{false}}.x^{false}.$$

Likewise we have, for instance,

$$Turn^A \stackrel{def}{=} asgn_{turn}^{A}.Turn^A + asgn_{turn}^{B}.Turn^B + \overline{n_{turn}^{A}}.Turn^A.$$

Peterson's mutual exclusion algorithm (PME) is the parallel composition of all these processes, restricting all the communications

$$(\mathrm{A} \mid \mathrm{B} \mid ReadyA^{false} \mid ReadyB^{false} \mid Turn^A)\backslash L,$$

where L is the set of all names except en_A, en_B, ln_A, ln_B, ec_A, ec_B, lc_A and lc_B [17].

The LTS of the above CCS expression PME is exactly as displayed in Fig. 6. By the interpretation of CCS as an LTSC, defined in Section 4.4, one obtains $\ell_2 \not\frown m_4$, where I use the names ℓ_2 and m_4 of the underlying instructions from Fig. 5 to denote the two outgoing τ-transitions from the state $_{\ell_2}{}^{A}{}_{m_4}$. In fact, this could have been concluded without studying the above CCS rendering of Peterson's protocol, as Section 4.4 remarks that in the LTSC of CCS the relation $\frown$ is symmetric, while Section 16 concludes that $m_4 \not\frown \ell_2$. Thus, the CCS rendering of Peterson's algorithm does not satisfy the correctness criterion EC$^\prime$.

20. What happens if processes try to read and write simultaneously

A program instruction like ℓ_2, ℓ_3, ℓ_4 or ℓ_7 that reads or writes a value *true*, *false*, *A* or *B* from or to a register *readyA*, *readyB* or *turn* cannot be executed instantaneously, and is thus assumed to occur during an interval of real time. Hence it may happen that Processes A and B try to access the same register during overlapping periods of time. In such a case is it common to assume that the register is *safe*, meaning that

> *A read operation not concurrent with any write operation returns the value written by the latest write operation, provided the last two write operations did not overlap.*

This assumption stems from [45], although overlapping writes were not considered there. "No assumption is made about the value obtained by a read that overlaps a write, except that it must obtain one of the possible values of the register." [45][20] In the same spirit, one may assume that two overlapping writes may put any of its possible values in the register, in the sense that subsequent reads will return that value. I will assume safety in this sense of the Boolean registers *readyA*, *readyB* and *turn*. In an architecture where safe registers are not available, and cannot be simulated, implementing a correct mutual exclusion protocol appears to be hopeless.[21]

For the fate of Peterson's algorithm it matters what happens if one process wants to start writing a register when another is busy reading it. There appear to be only three (or five) possibilities.

(1) The register cannot handle a read and a write at the same time; as the read started first, the writing process will need to await the termination of the read action before the write can commence.
(2) The register cannot handle a read and a write at the same time, but the write takes precedence and occurs when scheduled. This aborts the read action, which can restart after the write has terminated.
(3) The read and write proceed as scheduled, thus overlapping in time.

A fourth possibility could be that reads and writes are instantaneous after all, so that overlap can be avoided without postponing either. I deem this unrealistic and do not consider this option here. A potential fifth possibility could be a variation of (2), in which the read merely is interrupted, and resumes after the write is finished. In that case, as with option (3), it seems reasonable to assume that the read can return any value of the register.

In Dijkstra's original formulation of the mutual exclusion problem [16], possibility (1) above—*atomicity*—was assumed—see the quote in Section 9. Lamport, on the other hand, assumes (3) [42]. On his webpage https://lamport.azurewebsites.

[20] This is not really a restriction, for one can always follow a read action $r := x$ of a variable x, where r is a local register, by a default assignment $r := v_0$ in case the read yields a value that is out of range.

[21] This statement can be strengthened by considering its contrapositive: if one has a correct mutual exclusion protocol, safe registers can be simulated. Namely, when confronted with a weak memory, where it takes time for writes to propagate after the instruction has been encountered, one could put each read and write instruction in a critical section, together with an appropriate memory barrier or fence instruction, to ensure propagation of the write before any read occurs. This ought to yield safety.

net/pubs/pubs.html#bakery Lamport takes the position that assuming atomicity "cannot really be said to solve the mutual exclusion problem", as it assumes "lower-level mutual exclusion". As possibility (3) adds the complication of arbitrary register values returned by reads that overlap a write, he implicitly takes the position that solving the mutual exclusion problem under assumption (3) is more challenging; and this is exactly what his bakery algorithm does.

Here I argue that atomicity is the more challenging assumption. The objection that assuming reads and writes to be atomic amounts to assuming "lower-level mutual exclusion" is based on the idea that securing the mutex property of a mutual exclusion protocol is the main challenge. However, the real challenge is doing this in a starvation-free way, and this feature is not inherited from the lower level. By assuming atomicity one obtains $\ell_2 \not\frown m_4$, that is, transition ℓ_2 is affected by m_4, and consequently Peterson's algorithm fails the correctness requirement EC^J. In Section 22 below I will argue that this is not merely a result of the way I choose to model things in this paper, but actual evidence of the incorrectness of Peterson's algorithm, or any other mutual exclusion protocol for that matter, provided one consistently assumes atomicity, as well as speed independence.

Assuming (2) instead yields $\ell_2 \frown m_4$, that is, the write ℓ_2 is in no way affected by the read m_4. This means that nothing can prevent Process A from executing ℓ_2. This makes Peterson's algorithm correct, in the sense that it satisfies EC^J.

Assuming (3) also yields $\ell_2 \frown m_4$. Also this would make the algorithm correct, provided that it is robust against the effects of overlapping reads and writes. For Peterson's algorithm this is not the case; overlapping reads and writes can cause a violation of the mutex property ME—see Section 21. However, various other mutual exclusion protocols, including the ones of [42] and [2], are robust against the effects of overlapping reads and writes and do satisfy EC^J when assuming (3).[22]

In CCS, and in Petri nets, $\frown$ is symmetric, and one has $\ell \not\frown m$ whenever ℓ and m are read or write instructions on the same register, at least when employing the concurrency relation $\frown$ of Section 4.4. This amounts to assuming atomicity.

21. Is Peterson's protocol resistant against overlapping reads and writes?

Those mutual exclusion protocols that were designed to be robust under overlapping reads and writes, avoid overlapping writes altogether, either by making sure that each variable can be written by only one process (although it can be read by others) [42,58], or by putting writes to the same variable right before or after [2] the critical section, within the part of a process' cycle that is made mutually exclusive. Any protocol that doesn't take this precaution, including Peterson's, is regarded with suspicion by those that make an effort to avoid ill effects due to reads overlapping with writes. Nevertheless, until May 2021 no examples were known (to me at least) that overlapping reads and writes actually cause any problem for Peterson's protocol. I personally believed that it was robust under overlap, reasoning as follows [unpublished notes].

> *Two overlapping write actions to the same register may produce any value of that register. In Peterson's algorithm, the only register that can be written by both processes contains the variable turn. It is a Boolean register, whose values are A and B. The only write actions to this register are ℓ_3 and m_3. When these overlap in time, any of the register values, that is A or B, may result. However, ℓ_3 tries to write the value B, and m_3 the value A. So if the result of a simultaneous write is A, one can just as well assume that ℓ_3 occurred before m_3, and if it is B, that m_3 occurred before ℓ_3. Thus the effects of overlapping writes are no different than those of atomic writes, and hence harmless.*
>
> *Peterson's algorithm has six cases of a read overlapping with a write, and thanks to symmetry it suffices to study three of them. First consider the overlap of the write ℓ_2 with the read m_4. Here the overlapping read can yield any register value, that is, true or false. One should not ignore the possibility that Process* B*, while cycling around, performs multiple m_4-reads of readyA that may return any sequence of true and false during a single write action ℓ_2 of Process* A*. However, any read by Process* B *that returns readyA = true does not help to pass the* ***await****-statement in m_4, and is equivalent to no read of readyA being carried out. So all reads that matter return readyA = false, and can just as well be thought of as occurring prior to ℓ_2. A similar argument applies to the overlap of the write ℓ_3 with the read m_4, and of the write ℓ_7 with the read m_4; also here any resulting behaviour can already be generated without assuming an overlap.*

I leave it as a puzzle to the reader to find the fallacy in this argument.[23] A run of Peterson's algorithm that violates the mutex property ME was found in 2021 by means of the model checker mCRL2 by Myrthe Spronck [56]. This involved implementing safe registers, as described in Section 20, as mCRL2 processes.

22. The impossibility of mutual exclusion when assuming atomicity and speed independence

In this section I argue that when assuming atomicity and speed independence, Peterson's algorithm is not correct, in the sense that it fails the requirement of starvation-freedom. The argument is that the run corresponding with the path π_P from Section 16 can actually occur. To see why, let me illustrate the form of speed independence needed for this argument by means of a simple example in CCS.

[22] Szymański's algorithm [58] was designed specifically for this robustness, but fails to achieve it [57].

[23] On request of 2 referees I divulge this fallacy at http://theory.stanford.edu/~rvg/PMEfallacy.html—not here to avoid spoilers. Alternatively, one can find out by comparing with the counterexample in [56].

Example 22.1. Consider the process $(X|Y)\backslash\{c\}$ with $X \stackrel{def}{=} a.\mathbf{0} + \bar{c}.X$ and $Y \stackrel{def}{=} c.d.e.Y$, where none of the actions a, d, e is blocked by the environment, that is, the environment is continuously eager to partake in these actions. The question is whether action a is guaranteed to occur.

Here one may argue that when Process X reaches the state $a.\mathbf{0} + \bar{c}.X$ at a time when its environment, which is the Process Y, is not yet ready to engage in the synchronisation on c, it will proceed by executing a. If, on the other hand, both options a and $\bar{c}$ are available, it cannot be excluded that $\bar{c}$ is chosen, as no priority mechanism is at work here.

Now after execution of $\bar{c}|c$, Process X again faces a choice between a and $\bar{c}$, but Process Y first has to execute the actions d and e. During the time that Y is busy with d and e, for Process X it feels like $\bar{c}$ is blocked, and it will do a.

A strict interpretation of speed independence, which I employ here, says that actions d and e may be executed so fast that from the perspective of Process X one can just as well assume that no actions d and e were scheduled at all. Thus the answer to the above question will not change upon replacing Process Y by $Y' \stackrel{def}{=} c.Y'$. However, for the process $(X|Y')\backslash\{c\}$ there is really no reason to assume that a will ever occur.

In CCS and related process algebras this form of speed independence is built in: one sees d and e as τ-transitions, as for the purpose of answering the above question they can be regarded as unobservable, and then applies the law $a.\tau.P = a.P$ [47].

A run of Peterson's protocol, or any implementation thereof in a setting where atomicity and the above form of speed independence may be assumed, can visit the state $\ell_2^A m_4$ in which Process A wants to write on register *readyA*, and that register has a choice between being written or being read first, by Process B. One cannot exclude that the read action wins this race, which allows Process B to enter the critical section. During the execution of m_4, reading *readyA*, Process A has to wait. Afterwards, Process B might execute actions m_5–m_3 so fast that from the perspective of Process A no time elapses at all. This brings us again in State $\ell_2^A m_4$ where the same race between ℓ_2 and m_4 occurs. Again it could be won by m_4. This behaviour can continue indefinitely.

The above argument stems from [24], where it was made not just for Peterson's protocol, but for all conceivable mutual exclusion algorithms. These include Lamport's bakery algorithm [42], Szymański's algorithm [58], and the round-robin scheduler. To obtain a correct mutual exclusion algorithm, one has to either employ register hardware in which the assumption of atomicity—possibility (1) in Section 20—is not valid, or make the protocol speed-dependent. The first option was already explored in Section 20; as mentioned there, using hardware that works according to possibilities (2) or (3) solves the problem. The second option will be explored in Part IV of this paper.

23. Variations of Petri nets and CCS with non-blocking reading

To escape from the failure of requirement EC^J for Peterson's protocol due to the assumption (1) of atomicity as well as speed independence, one can instead assume possibility (2) from Section 20 while keeping speed independence. I refer to such an option as *non-blocking reading* [12], as a write cannot be postponed by a read action on the same register. This yields $\ell_2 \smile\!\!\bullet\, m_4$, thereby saving EC^J. Here I review how this can be modelled in variations of Petri nets and CCS.

23.1. Read arcs

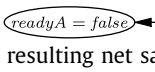
A Petri net with *read arcs* is a tuple $N = (S, T, F, R, M_0, \ell)$ as in Definition 4.3, but enriched with an object $R: S \times T \to \mathbb{N}$, such that $F(s,t) > 0 \Rightarrow R(s,t) = 0$. An element (s,t) of the multiset R is called a *read arc*. Read arcs are drawn as lines withoutarrow heads. For $t \in T$, the multiset $\hat{t}: S \to \mathbb{N}$ is given by $\hat{t}(s) = R(s,t)$ for all $s \in S$. Transition t is *enabled* under M iff ${}^\bullet t + \hat{t} \leq M$. In that case $M \xrightarrow{t}_N M'$, where $M' = (M - {}^\bullet t) + t^\bullet$.

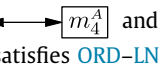
Thus, for a transition to be enabled, there need to be enough tokens at the other end of each of its read arcs. However, these tokens are not consumed by the firing. Clearly, the transition relation $\xrightarrow{t}_N$ between the markings of a net is unaffected when replacing each read arc (s,t) by a loop between s and t; that is, by dropping R and using the flow relation $F_R := F + R + R^{-1}$. This does not apply to the concurrency relation between transitions.

The definition of a structural conflict net can be extended to Petri nets with read arcs by requiring, for all $t, u \in T$ and all reachable markings M, that

$$\text{if } ({}^\bullet t + \hat{t}) \cap {}^\bullet u \neq \emptyset \text{ then } {}^\bullet t + \hat{t} + {}^\bullet u \not\leq M \text{ or } \hat{u} \not\leq M.$$

For such nets, one defines $t \smile\!\!\bullet\, u$ iff $({}^\bullet t + \hat{t}) \cap {}^\bullet u = \emptyset$. This says that a transition u affects a transition t iff u consumes a token that is needed to enable t. The condition for a structural conflict net guarantees exactly that if $t \not\smile\!\!\bullet\, u$ and u is enabled under a reachable marking M, then t is not enabled under the marking $M - {}^\bullet u$.

As pointed out by Vogler in [60], the addition of read arcs makes Petri nets sufficiently expressive to model mutual exclusion. When changing the loops 

and 

in Fig. 7 into read arcs, one obtains $\ell_2 \smile\!\!\bullet\, m_4^B$ and $m_2 \smile\!\!\bullet\, \ell_4^A$. So the resulting net satisfies ORD–LN with $CC = J$, and correctly solves the mutual exclusion problem. Two different solutions as Petri nets with read arcs are given in [60, Figures 8 and 9], the one in Figure 9 being a round-robin scheduler.

23.2. Broadcast communication

A process algebraic solution was presented in [31, Section 5], using an extension ABC of CCS with broadcast communication. The most obvious distinction between broadcast communication and CCS-style handshaking communication is that the former allows multiple recipients of a message and the latter exactly one. This feature of broadcast communication was not exploited in the solution of [31]. A more subtle feature of broadcast communication is that the transmission of a broadcast occurs regardless of whether anyone is listening. Thus a broadcast can be used to model a write that cannot be blocked or postponed because the receiving register is busy being read. This yields an asymmetric concurrency relation, in which a broadcast transition is not affected by a competing transition from a receiver of the broadcast, whereas the competing transition *is* affected by the broadcast.

Nevertheless, the semantics of ABC says that the broadcast *will* be received by all processes that are in a state with an outgoing receive transition. This allows one to make receipt of a broadcast reliable, by giving the receiving process an outgoing receive action in each of its states. This feature of the semantics of ABC, which is essential for modelling mutual exclusion, is somewhat debatable, as one could argue that a process that is engaged in its own broadcast transmission through a transition between two states that each have an outgoing receive transition, is temporary too busy to hear an incoming message.

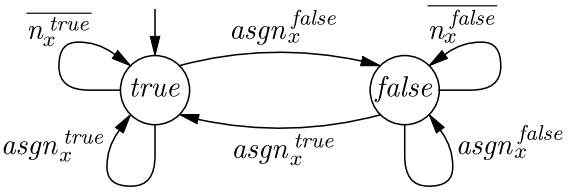
The solution proposed in [31] is not a mutual exclusion protocol, but a fair scheduler, which can be converted into a mutual exclusion protocol in the manner of Section 10. In fact, it is a variant of the encapsulated gatekeeper, where broadcast communication is used to merely assume justness for all requirements. In the same spirit one can model Peterson's protocol in ABC in such a way that EC^{J} is satisfied. It suffers to interpret $\overline{asgn_x^v}$ and $asgn_x^v$ as broadcast transmit and receive actions. The LTSC semantics of ABC [25] then yields $\ell_2 \smile\!\!\bullet\, m_4$.

23.3. Signals

A different process algebraic solution, arguably less debatable, was proposed in [12,17]. In [12], processes P are equipped with the possibility to perform actions that do not change their state, and that, in synchronisation with another parallel process Q, describe information on the state of P that is read by Q in a non-blocking way. In [17], following [4], such actions are called *signals*. The communication between, say, a traffic light, emitting the signal *red*, and a car, coming to a halt, is binary, just like handshaking communication in CCS. The difference is that the concurrency relation between transitions again becomes asymmetric, because the car is affected by the traffic light, but the traffic light is not affected by the car. A car stopping for a red light in no way blocks or postpones the action of the traffic light of turning green.

In [4,17] the emission of a signal is modelled as a predicate on states, whereas the receipt of such an emission is modelled as a transition. In [12,6] the emission of a signal is modelled as a transition instead. An advantage of the former approach is that it stresses the semantic difference between signal emissions and handshaking actions, and emphasises that signal emissions cannot possibly cause a state-change. An advantage of the latter approach is that communication via signals can be treated in the same way as a CCS handshaking communication, thereby simplifying the process algebra. Technically, the two approaches are equivalent.

In CCS with signals [17], modelling signal emissions as transitions [6,25], a Boolean variable x, such as *readyA* in Peterson's protocol, has the exact same LTS as in the CCS model of Section 19: However, this time the notifications $\overline{n_x^v}$ are signals emissions rather than handshaking actions. The definition of the concurrency relation $\smile\!\!\bullet$ on CCS transitions from Section 4.4 is in [25] extended to CCS with signals in such a way that with the above way of modelling variables, and the same processes A and B as in Section 19, one obtains $\ell_2 \smile\!\!\bullet\, m_4$. Here the action m_4 of reading the register is like a car reading a traffic light, and does not inhibit the write action ℓ_2 on the same register.



This shows that Peterson's algorithm can be correctly modelled in CCS with signals. Earlier, Dekker's algorithm was correctly modelled in the process algebra PAFAS with non-blocking reading [12], and same was done for Peterson's algorithm in [10]. The latter paper also points out that non-blocking reading is not strong enough an assumption to obtain starvation-freedom, or even deadlock-freedom as defined in Section 9, for Knuth's algorithm [41]; this requires a fairness assumption.

23.4. Modelling non-blocking reading in CCS

The rendering in [17] of Peterson's protocol employs an extension of CCS with signals that arguably strictly increases the expressiveness of the language. Namely in CCS with signals one obtains an asymmetric concurrency relation $\smile\!\!\bullet$, which turned out to be essential for the satisfactory modelling of Peterson; restricted to proper CCS this relation is symmetric. Bouwman [6] proposes the same modelling of Peterson's protocol, but entirely within the confines of the existing language CCS, namely by simply declaring some of the action names of CCS to be signals. As it is essential that the emission of a signal never causes a state-change, care has to be taken to only use CCS-expressions in which the actions that are chosen to be

seen as signal emissions occur in self-loops only. When doing this, creating a satisfactory rendering of Peterson's protocol within CCS is unproblematic [6]. Neither [17] nor [6] mention the concurrency relation $\frown\!\!\!\!\bullet$ at all, and use a coinductive definition of justness instead. However, as shown in [25], the concept of justness common to [17] and [6] can equivalently be obtained as in Section 5 from an asymmetric concurrency relation $\frown\!\!\!\!\bullet$. Thus, by declaring certain CCS actions to be signals, Bouwman effectively changes the concurrency relation between CCS transitions labelled with those actions.

In [32] it was stated that fair schedulers and mutual exclusion protocols cannot be rendered correctly in CCS without imposing a fairness assumption. In making that statement, the (symmetric) concurrency relation $\frown$ between CCS transitions defined in Section 4.4—or equivalently, the resulting notion of justness—was seen as an integral part of the semantics of CCS. In the early days of CCS, a lot of work has been done in formalising notions of concurrency for CCS and related process algebras [50,35,5,62,34,14,51]. All that work is consistent with the concurrency relation $\frown$ defined in Section 4.4. Changing the concurrency relation, as implicitly proposed by Bouwman, alters the language CCS as seen from the perspective of [50,35,5,62,34,14,51]. However, it is entirely consistent with the interleaving semantics of CCS given by Milner [47].

23.5. Modelling and verification of Peterson's algorithm with mCRL2

ACP [3,19] and mCRL2 [36] are CCS-like process algebras that fall under the scope of the impossibility result of [32]. That is, when defining a concurrency relation on the transitions of ACP or mCRL2 processes in the traditional way, consistent with the viewpoint of [50,35,5,62,34,14,51], and defining justness as in Section 5 in terms of this concurrency relation, Peterson's algorithm cannot be correctly rendered in ACP or mCRL2 when merely assuming justness, i.e., without resorting to a fairness assumption. Nevertheless, by the argument of [6], Peterson's algorithm *can* be correctly rendered in these formalisms under the assumption of justness when using an alternative concurrency relation, one obtained by treating certain actions as signals.

Using this insight, Bouwman, Luttik and Willemse [7] render Peterson's algorithm in an instance of ACP or mCRL2 that uses much more informative actions. This could be done without adapting those languages in any way, simply by choosing appropriate actions. Each action labelling a transition contains additional information on the components of the system from which this transition stems. This information is preserved under synchronisation, when a transition of a parallel composition is built from transitions of each of the components. The latter requires the general communication format of ACP or mCRL2, as in CCS synchronisation merely result in τ-transitions. A price paid for this approach is that the resulting LTSs have much fewer τ-transitions, so that there are fewer opportunities for state-space reduction by abstraction from internal activity.

In Section 7 I showed how B-justness, and thereby the correctness criteria for mutual exclusion protocols, can be expressed in standard LTL enriched with a number of atomic propositions, such as en^t and $\sharp t$. Bouwman, Luttik and Willemse [7] show not only that the same can be done in the modal μ-calculus, but also that all the necessary atomic propositions can be expressed in terms of the carefully chosen actions that are used in modelling the protocol. This made it possible to model the protocol in such way that all correctness requirements can be checked with the existing mCRL2 toolset [9].

Part IV. A speed-dependent rendering of Peterson's protocol

In Part III I showed that when assuming atomicity and speed independence, Peterson's algorithm does not have the correctness property EC^J—and in [24] I argued that the same can be said for any other mutual exclusion protocol. That is, it satisfies starvation-freedom only under a weak fairness assumption, which in my opinion is not warranted — if it was, even the encapsulated gatekeeper of Section 15 would be an acceptable mutual exclusion protocol.

Thus, to obtain a correct version of Peterson's algorithm (or mutual exclusion in general) one has to either drop the assumption of atomicity, or speed-independence. The former possibility has been elaborated in Part III; Section 16 showed then when assuming non-blocking reading (option (2) from Section 20, resulting in the verdict $\ell_2 \frown\!\!\!\!\bullet\, m_4$) instead of atomicity, Peterson's algorithm is entirely correct. Section 23 recalled how to model this with process algebra or Petri nets.

The latter possibility will be elaborated here. I present a speed-dependent incarnation of Peterson's protocol that satisfies all correctness requirements, even under the assumption of atomicity. Moreover, in the model of Section 26, requirement EC^J can be strengthened into EC^{Pr}, thereby attaining the best quality criteria in the hierarchy of Fig. 3. As pointed out in Part III, this is not possible when keeping speed independence, even when dropping atomicity.

The idea is extremely simple. As explained in Section 16, all that is needed to obtain starvation-freedom is the certainty that when Process A reaches the state ℓ_2, it will in fact execute the instruction ℓ_2. The only thing that can stop Process A in state ℓ_2 from executing ℓ_2, is the register *readyA* being too busy being read by Process B to find time for being writtenby Process A. Now assume that there exists an amount t_0 of time, such that, if for a period of at least t_0, Process A is in state ℓ_2 and the register *readyA* is available, in the sense that it is not being read by Process B, then in that period the write action ℓ_2 will commence. Now further assume that Process B will spend a period of at least t_0 in its critical or in its noncritical section. Then Process A will have enough time to perform the action $readyA := true$, and starvation-freedom is ensured. This is the speed-dependent version of Peterson's protocol I propose. In fact I cannot exclude that mutual exclusion protocols work in practice exactly because timing constraints such as sketched above are always met.

The above solution is sufficiently clear not to need mathematical proof. Nevertheless I proceed with an implementation of the above idea in process algebra. The goal of this is mostly to see which process algebra we need to formalise time-dependent reasoning such as performed above. Naturally a timed process algebra that associates real numbers to various

passages of time would be entirely equipped for this task. However, I will show that the idea can already be formalised in a realm of untimed process algebra, in the sense that the progress of time is not quantified.

24. CCS with time-outs

Following [29,27], my process algebra will be $\mathrm{CCS_t}$, which is CCS, as presented in Section 4.3, but with α ranging over $Act := \mathscr{A} \dot\cup \bar{\mathscr{A}} \dot\cup \{\tau, \mathrm{t}\}$, with t a fresh *time-out action*. Relabellings f extend to this extended set of actions Act by $f(\mathrm{t}) := \mathrm{t}$. The interpretation of this language as an LTSC proceeds exactly as in Section 4.4.

All actions $\alpha \in Act$ are assumed to occur instantaneously. The time-out action t models the end of a time-consuming activity from which we abstract. When a system arrives in a state P, and at that time X is the set of actions allowed (= not blocked) by the environment, there are two possibilities. If P has an outgoing transition u with $\ell(u) \in X \cup \{\tau\}$, the system immediately takes one of the outgoing transitions u with $\ell(u) \in X \cup \{\tau\}$, without spending any time in state P. The choice between these transitions is entirely nondeterministic. The system cannot immediately take a transition u with $\ell(u) \in A \backslash X$, because the action $\ell(u)$ is blocked by the environment. Neither can it immediately take a transition u with $\ell(u) = \mathrm{t}$, because such transitions model the end of an activity with a finite but positive duration that started when reaching state P.

In case P has no outgoing transition u with $\ell(u) \in X \cup \{\tau\}$, the system idles in state P for a positive amount of time. This idling can end in two possible ways. Either one of the time-out transitions $P \xrightarrow{\mathrm{t}} Q$ occurs, or the environment spontaneously changes the set of actions it allows into a different set Y with the property that $P \xrightarrow{a} Q$ for some $a \in Y$. In the latter case a transition u with $src(u) = P$ and $\ell(u) = a \in Y$ occurs. The choice between the various ways to end a period of idling is entirely nondeterministic. It is possible to stay forever in state P only if there are no outgoing time-out transitions.[24]

A fundamental law describing the interaction between τ- and t-transitions, motivated by the above, is $\tau.P + \mathrm{t}.Q = \tau.P$. It says that when faced with a choice between a τ- and a t-transition, a system will never take the t-transition. I could have devised an operational semantics of $\mathrm{CCS_t}$, featuring negative premises, that suppresses the generation of transitions $R \xrightarrow{\mathrm{t}, C} Q$ when there is a transition $R \xrightarrow{\tau, D} P$. However, following [29,27], I take a different, and simpler, approach. The operational semantics of $\mathrm{CCS_t}$ is exactly like the one of CCS, and generates such spurious transitions $R \xrightarrow{\mathrm{t}, C} Q$; instead, its semantics assures that these transitions are never taken. In [27] a branching time semantics is proposed, and in [29] a linear-time semantics—the closest approximation of partial trace semantics [23] that yields a congruence for the operators of $\mathrm{CCS_t}$. Both these semantics satisfy $\tau.P + \mathrm{t}.Q = \tau.P$.

Here I achieve the same by calling a path *potentially complete* when it features no transitions $R \xrightarrow{\mathrm{t}, C} Q$ when there also exists a transition $R \xrightarrow{\tau, D} P$. A completeness criterion now should set apart a subset of the potentially complete paths as being complete. So all paths containing spurious transitions $R \xrightarrow{\mathrm{t}, C} Q$ count as incomplete, and hence do not contribute to the evaluation of judgements in reactive temporal logic. In depictions of LTSs for fragments of $\mathrm{CCS_t}$ I will display the spurious transitions dotted, to emphasise that they cannot be taken.

A transition $R \xrightarrow{\mathrm{t}, C} Q$ also cannot be taken when there is an alternative $R \xrightarrow{a, D} P$, with a an action that surely will not be blocked by the environment when the system is in state R. Thus, whether or not a transition is spurious depends on the mood of the environment at the time this transition is enabled. This dependency is encoded in the semantic equivalences of [29] and [27]. Given this, it was no extra effort to simultaneously inhibit the selection of transitions that are spurious in any environment.

25. Spurious transitions and completeness criteria for LTSs with time-outs

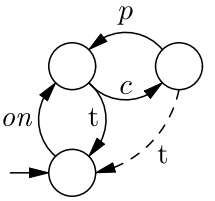
In LTSs with t-transitions, it makes sense to allow judgements $P \models^{CC}_{B,E} \varphi$ with $B \subseteq E \subseteq A$, where A is the set of all actions except τ and t. Here B is the set of actions that can be permanently blocked by the environment, and E the ones that can be blocked for finite periods of time. Thus, the annotations B and E rule out those environments in which an action from $A \backslash B$ is blocked permanently, or an action from $A \backslash E$ is blocked temporarily. My interest is in the cases $CC = Pr$ and $CC = J$.

Definition 25.1. A transition u is *E-spurious* if $\ell(u) = \mathrm{t}$ and there exists a transition $v \in Tr$ with $src(v) = src(u)$ and $\ell(v) \in (A \backslash E) \cup \{\tau\}$. It is *spurious* iff it is A-spurious.

[24] The environment in which a $\mathrm{CCS_t}$ process P runs can be anthropomorphised as a user behind a switchboard who can toggle each visible action as "blocked" or "allowed" [23, Section 5]. The user can toggle switches either as an immediate response on a visible action performed by P, or at arbitrary points in time. Alternatively, such an environment can be seen as a $\mathrm{CCS_t}$ process E, that synchronously runs in parallel with P, yielding the environment/system composition $(E|P)\backslash A$. As an example, take $P = \mathrm{t}.c.\mathbf{0} + b.\mathbf{0}$ and $E = \mathrm{t}.\bar{b}.\mathbf{0} + \bar{c}.\mathbf{0}$. This environment first allows only c to occur, but after some time allows b (while blocking c). The composition $(E|P)\backslash A$ starts by idling, and then performs a τ-transition, that from the perspective of P is either c or b. Which one depends on which of the two time-outs, from P or E, occurs first.

Note that u is spurious iff it is E-spurious for all E. This is the case iff it is a t-transition sharing its source state with a τ-transition. As actions from $A\backslash E$ cannot be blocked by the environment, not even temporarily, E-spurious transitions cannot be taken.

Example 25.2. Consider the variant of the vending machine from Section 2 that turns itself off after some period of inactivity. This is modelled by the two time-out transitions on the right. The machine also has an *on* transition, to be used by its operator to start it up. The dashedt-transition is $\{c, on\}$-spurious: it cannot be taken in an environment where a user of the machine never blocks the production of a pretzel.



Definition 25.3. A path π is *potentially E-complete* if it contains no E-spurious transitions. It is (B, E)-*progressing* if it (a) is potentially E-complete, and (b) is either infinite or ends in a state of which all outgoing transitions have a label from B. It is (B, E)-*just* if (a) it is potentially E-complete, and (b) for each $u \in Tr$ with $\ell(u) \notin B$ and whose source state $s := src(u)$ occurs in π, any suffix of π starting at s contains a transition v with $u \leadsto v$.

The judgement $s \models^{Pr}_{B,E} \varphi$ (resp. $s \models^{J}_{B,E} \varphi$) holds if $\pi \models \varphi$ holds for all (B, E)-progressing (resp. (B, E)-just) paths π starting in s.

For a finite path to be complete, its last state may have outgoing transitions with labels from B only, for a run comes to an end only when all subsequent activity is permanently blocked by the environment. In the absence of t-transitions, judgements $s \models^{CC}_{B,E} \varphi$ are independent of E, and agree with the ones defined in Part I of this paper.

Example 25.4. In Example 25.2 one has $\leadsto = \emptyset$, so (B, E)-just is the same as (B, E)-progressing. Let $B = \{c, on\}$. Each (B, B)-just path contains as many p- as c-transitions (possibly ∞). However, a (B, A)-just path may contain strictly more c-transitions.

In the context of the present paper, when describing properties for a given or desired process P, I see no reason to combine judgements $P \models^{CC}_{B,E} \varphi$ with different values of E. This suggests writing $P \models^{CC}_{B,E} \varphi$ as $(P, E) \models^{CC}_{B} \varphi$. This way, the quality criteria of Sections 11 and 12 can remain unchanged, and apply to systems (P, E). Here P is a hypothetical fair scheduler or mutual exclusion protocol, and E the set of its actions that can be temporarily blocked by the environment.

To gauge the influence of the environment on the visible actions en_i, ln_i, ec_i and lc_i of a mutual exclusion protocol, one can see the processes i that compete for the critical section as clients that communicate with the protocol through synchronisation on these actions. As explained in Section 12, the actions ln_i belong in B (except when formulating requirement LN) because ln_i is permanently blocked in case Client i chooses not to leave its noncritical section again. The actions lc_i belong in E, but not in B, because the client may need some time before leaving its critical section, but is assumed to do this eventually. As mentioned in Section 12, the actions ec_i and en_i do not belong in B, for we assume the client to eventually enter the (non)critical section when allowed by the protocol. There is a choice between putting these actions in E or not. Putting them in E models that the client may delay a while before entering the (non)critical section when allowed, whereas putting them in $A\backslash E$ models that when the protocol for Client i is in its *entry* or *exit* section, the actual client will patiently wait until granted access to the (non)critical section, and take advantage of this opportunity as soon as it arises. Taking $E = E_l := \{ln_i, lc_i \mid i = 1, \ldots, N\}$ appears most natural, but taking $E = A = \{ln_i, ec_i, lc_i, en_i \mid i = 1, \ldots, N\}$ is a reasonable alternative. The latter leads to stronger judgements, in the sense that when a protocol P is correct when taking $E := A$, it is surely correct when taking $E := E_l$. To model both options at the same time, in Fig. 8 I will draw E_l-spurious transitions dashed. Those transitions cannot be taken when choosing $E := E_l$, but they can be taken when choosing $E := A$.

26. Modelling Peterson's protocol in CCS with timeouts

My model of Peterson's algorithm in CCS_t differs from the one from Section 19 in only one way: a t-action is inserted between ec_i and lc_i for $i = \text{A}, \text{B}$, in the two processes A and B. Thus $\text{A} \stackrel{def}{=} ln_A \,.\, \overline{asgn^{true}_{readyA}} \,.\, \overline{asgn^{B}_{turn}} \,.\, (n^{false}_{readyB} + n^{A}_{turn}) \,.\, ec_A \,.\, \text{t} \,.\, lc_A \,.\, \overline{asgn^{false}_{readyA}} \,.\, en_A \,.\, \text{A}$.

This models that a process spends a positive but finite amount of time in its critical section. The LTS of the resulting CCS_t rendering of Peterson's protocol is displayed in Fig. 8. Exactly as in Section 16/19, it follows that this model satisfies the requirements ORD, ME, LC^{Pr}, EN^{J} and LN^{J}. Additionally, it satisfies EC^{Pr}, as follows immediately from the LTS.

The same result would be obtained by letting time pass in the noncritical section, instead of, or in addition to, the critical section. It can be argued that it is not realistic to assume that assignments like ℓ_2 and ℓ_3 occur instantaneously. However, this part of the modelling in CCS_t is merely an abstraction, and can be taken to mean that the time needed to execute such an assignment is significantly smaller than the time a process spends in its critical and/or noncritical section. Using CCS_t, one can also make a model in which time is spent between each two instructions. In such a rendering one would obtain EC^{J}, thus needing justness for starvation-freedom.

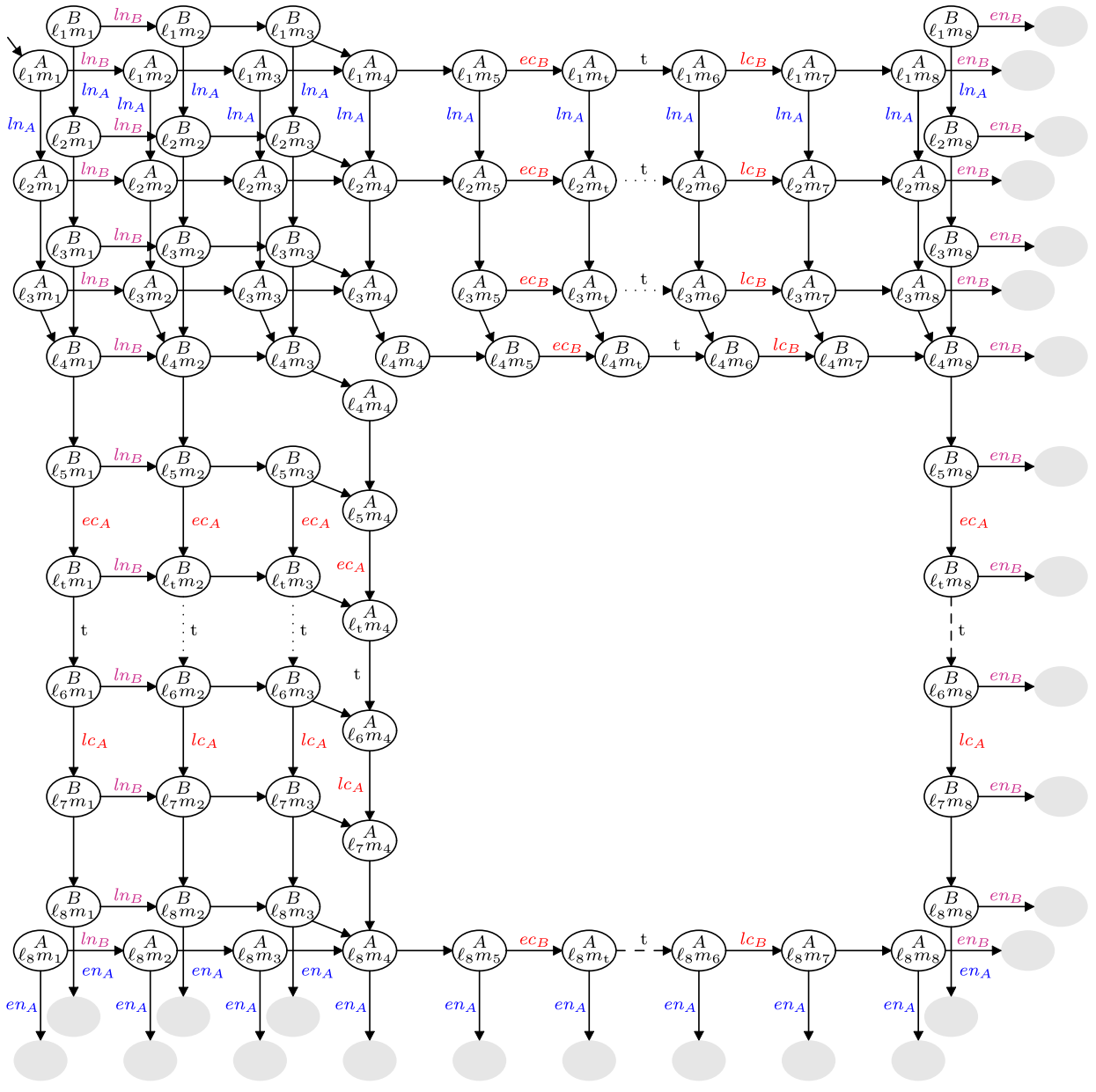


Fig. 8. Speed dependent LTS of Peterson's mutual exclusion algorithm.

27. Conclusion

This paper introduces temporal judgements of the form $\mathscr{D} \models^{CC}_{B,E} \varphi$, where

- $\mathscr{D}$ is a distributed system or its representation as pseudocode, a Petri net, a process in some process algebra, or a state in a labelled transition system or in a Kripke structure,
- φ is a formula from a temporal logic, such as LTL or CTL,
- CC is a completeness criterion, telling which execution paths model actual system runs,
- and B and E model the influence of the environment on reactive system behaviour, by stipulating which actions can be blocked permanently and temporary, respectively.

I call this *reactive temporal logic*, or *reactive LTL* when φ is an LTL formula. Standard temporal logic has judgements $\mathscr{D} \models \varphi$, obtained by default choices for CC, B and E.

In the absence of *time-out* transitions, the truth of judgements $\mathscr{D} \models^{CC}_{B,E} \varphi$ is independent of E, so that $\models$ reduces to a quaternary relation. In this context I present encodings of reactive LTL into standard LTL, at the expense of adding many atomic propositions, and I present a fragment of LTL that only describes *safety properties*, telling that nothing bad will ever happen. On this fragment, the values of CC and B do not matter, so that reactive temporal judgements say no more than standard ones.

I formulate the correctness requirements of mutual exclusion protocols and fair schedules in reactive LTL, so that it is unambiguously determined which processes present correct mutual exclusion protocols, or fair schedules, and which do not. As some of the criteria are parametrised by the choice of a completeness criterion, I obtain a hierarchy of correctness requirements, where the choice of a stronger completeness criterion yields a lower quality mutual exclusion protocol or fair scheduler.

I formulate two assumptions that are commonly made when studying mutual exclusion, and call them *atomicity* and *speed independence*. Both stem from Dijkstra's paper in which the mutual exclusion problem was originally posed. I claim that under these assumptions correct mutual exclusion protocols do not exist, unless one accepts the lowest quality criteria from the above-mentioned hierarchy, namely the choice of *fairness* as completeness criterion. I substantiate this claim in detail for Peterson's mutual exclusion protocol. I consider the choice of fairness as unwarranted, because the real world is not fair. Moreover, when fairness would be an acceptable choice I propose a much simpler mutual exclusion protocol—the *gatekeeper*—that would also be correct, but which I expect would be rejected by most experts, exactly because its blatant employ of fairness.

I render Peterson's protocol as a Petri net and as an expression in the process algebra CCS. Since both atomicity and speed independence are build in in these formalisms, it is unavoidable that the so formalised protocol is correct only under the assumption of fairness.

Good requirements for mutual exclusion protocols are obtained by using *justness* as parameter in the above hierarchy of quality criteria. Justness is a completeness criteria that is weaker than fairness, and typically warranted in applications. Justness can be formalised in terms of a concurrency relation between transitions in labelled transition systems or Petri nets. I use Peterson's protocol to illustrate that any speed independent formalisation of mutual exclusion (implicitly or explicitly) requires an asymmetric concurrency relation.

Progress is a completeness criteria even weaker than justness, so that its use as parameter in the correctness criteria specifies even higher quality mutual exclusion protocols. I claim that such protocols do not exist when assuming speed independence, even when dropping the assumption of atomicity. Also this claim is substantiated in detail for Peterson's protocol.

One alternative for atomicity is to allow read and write actions to overlap in time. Assuming that two overlapping writes can write any legal value in a register, and a read overlapping with a write may read any legal value, Lamport's bakery algorithm [42] and Aravind's mutual exclusion algorithm [2] are known to work correctly: they satisfy all my requirements with justness as parameter. However, the algorithms of Peterson [52] and Szymański's [58] do not [56,57].

Another alternative to atomicity is to let a write action interrupt a read. This yields an entirely correct model of Peterson's algorithm, satisfying all requirements with justness as parameter. This can be modelled, for instance, in terms of Petri nets extended with read arcs [60], or CCS extended with signals [12,17]. These approaches yield an asymmetric concurrency relation.

Here I present a correct rendering of Peterson's algorithm that assumes atomicity and a symmetric concurrency relation. It satisfies all requirements with justness as parameter, and the main starvation-freedom requirement even with progress as parameter. Naturally, this goes at the expense of speed independence. I formalise this in a variant of the process algebra CCS enriched with *time-out* transitions. These allow to model the passage of time in a qualitative way, abstracting from exact durations.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

No data was used for the research described in the article.

Acknowledgments

I am grateful to Wan Fokkink, Peter Höfner, Bas Luttik, Liam O'Connor, Myrthe Spronck, Walter Vogler, Weiyou Wang and three reviewers for insightful feedback.

References

[1] K.R. Apt, N. Francez, S. Katz, Appraising fairness in languages for distributed programming, Distrib. Comput. 2 (4) (1988) 226–241, https://doi.org/10.1007/BF01872848.

[2] A.A. Aravind, Yet another simple solution for the concurrent programming control problem, IEEE Trans. Parallel Distrib. Syst. 22 (6) (2011) 1056–1063, https://doi.org/10.1109/TPDS.2010.172.
[3] J.C.M. Baeten, W.P. Weijland, Process Algebra, Cambridge Tracts in Theoretical Computer Science, vol. 18, Cambridge University Press, 1990.
[4] J.A. Bergstra, ACP with signals, in: J. Grabowski, P. Lescanne, W. Wechler (Eds.), Proc. International Workshop on Algebraic and Logic Programming, in: LNCS, vol. 343, Springer, 1988, pp. 11–20.
[5] G. Boudol, I. Castellani, On the semantics of concurrency: partial orders and transition systems, in: H. Ehrig, R. Kowalski, G. Levi, U. Montanari (Eds.), Proc. TAPSOFT'87, vol. 1, in: LNCS, vol. 249, Springer, 1987, pp. 123–137.
[6] M.S. Bouwman, Liveness analysis in process algebra: simpler techniques to model mutex algorithms, Technical Report, Eindhoven University of Technology, 2018, Available at http://www.win.tue.nl/~timw/downloads/bouwman_seminar.pdf.
[7] M.S. Bouwman, B. Luttik, T.A.C. Willemse, Off-the-shelf automated analysis of liveness properties for just paths, Acta Inform. 57 (3–5) (2020) 551–590, https://doi.org/10.1007/s00236-020-00371-w.
[8] S.D. Brookes, C.A.R. Hoare, A.W. Roscoe, A theory of communicating sequential processes, J. ACM 31 (3) (1984) 560–599, https://doi.org/10.1145/828.833.
[9] O. Bunte, J.F. Groote, J.J.A. Keiren, M. Laveaux, T. Neele, E.P. de Vink, W. Wesselink, A. Wijs, T.A.C. Willemse, The mCRL2 toolset for analysing concurrent systems—improvements in expressivity and usability, in: T. Vojnar, L. Zhang (Eds.), Proc. 25th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS'19, held as part of the European Joint Conferences on Theory and Practice of Software, ETAPS'19, Prague, Czech Republic, in: LNCS, vol. 11428, Springer, 2019, pp. 21–39.
[10] F. Buti, M. Callisto De Donato, F. Corradini, M.R. Di Berardini, W. Vogler, Automated analysis of MUTEX algorithms with FASE, in: G. D'Agostino, S. La Torre (Eds.), Proc. GandALF'11, Electronic Proceedings in Computer Science, vol. 54, Open Publishing Association, 2011, pp. 45–59.
[11] F. Corradini, M.R. Di Berardini, W. Vogler, Fairness of actions in system computations, Artif. Intell. 43 (2) (2006) 73–130, https://doi.org/10.1007/s00236-006-0011-2.
[12] F. Corradini, M.R. Di Berardini, W. Vogler, Time and fairness in a process algebra with non-blocking reading, in: M. Nielsen, A. Kucera, P. Bro Miltersen, C. Palamidessi, P. Tuma, F.D. Valencia (Eds.), Theory and Practice of Computer Science, SOFSEM'09, in: LNCS, vol. 5404, Springer, 2009, pp. 193–204.
[13] R. De Nicola, F.W. Vaandrager, Three logics for branching bisimulation, J. ACM 42 (2) (1995) 458–487, https://doi.org/10.1145/201019.201032.
[14] P. Degano, R. De Nicola, U. Montanari, CCS is an (augmented) contact free C/E system, in: M. Venturini Zilli (Ed.), Advanced School on Mathematical Models for the Semantics of Parallelism, 1986, in: LNCS, vol. 280, Springer, 1987, pp. 144–165.
[15] E.W. Dijkstra, (1962 or 1963): Over de sequentialiteit van processbeschrijvingen, Available at http://www.cs.utexas.edu/users/EWD/ewd00xx/EWD35.PDF.
[16] E.W. Dijkstra, Solution of a problem in concurrent programming control, Commun. ACM 8 (9) (1965) 569, https://doi.org/10.1145/365559.365617.
[17] V. Dyseryn, R.J. van Glabbeek, P. Höfner, Analysing mutual exclusion using process algebra with signals, in: K. Peters, S. Tini (Eds.), Proc. Combined 24th International Workshop on Expressiveness in Concurrency and 14th Workshop on Structural Operational Semantics, Berlin, Germany, in: Electronic Proceedings in Theoretical Computer Science, vol. 255, Open Publishing Association, 2017, pp. 18–34.
[18] J. Esparza, G. Bruns, Trapping mutual exclusion in the box calculus, Theor. Comput. Sci. 153 (1–2) (1996) 95–128, https://doi.org/10.1016/0304-3975(95)00119-0.
[19] W.J. Fokkink, Introduction to Process Algebra, Texts in Theoretical Computer Science, an EATCS Series, Springer, 2000.
[20] N. Francez, Fairness, Springer, New York, 1986.
[21] D.M. Gabbay, A. Pnueli, S. Shelah, J. Stavi, On the temporal analysis of fairness, in: P.W. Abrahams, R.J. Lipton, S.R. Bourne (Eds.), Proc. POPL '80, ACM Press, 1980, pp. 163–173.
[22] R.J. van Glabbeek, The linear time – branching time spectrum II; the semantics of sequential systems with silent moves, in: E. Best (Ed.), Proc. CONCUR'93, 4th International Conference on Concurrency Theory, Hildesheim, Germany, in: LNCS, vol. 715, Springer, 1993, pp. 66–81.
[23] R.J. van Glabbeek, The linear time – branching time spectrum I; the semantics of concrete, sequential processes, in: J.A. Bergstra, A. Ponse, S.A. Smolka (Eds.), Handbook of Process Algebra, Elsevier, 2001, pp. 3–99, chapter 1.
[24] R.J. van Glabbeek, Is speed-independent mutual exclusion implementable?, in: S. Schewe, L. Zhang (Eds.), Proc. 29th International Conference on Concurrency Theory, CONCUR'18, Beijing, China, in: Leibniz International Proceedings in Informatics (LIPIcs), vol. 118, Schloss Dagstuhl–Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 2018.
[25] R.J. van Glabbeek, Justness: a completeness criterion for capturing liveness properties (extended abstract), in: M. Bojańczyk, A. Simpson (Eds.), Proc. 22nd International Conference on Foundations of Software Science and Computation Structures, FoSSaCS'19; held as part of the European Joint Conferences on Theory and Practice of Software, ETAPS'19, Prague, Czech Republic, in: LNCS, vol. 11425, Springer, 2019, pp. 505–522.
[26] R.J. van Glabbeek, Ensuring liveness properties of distributed systems: open problems, J. Log. Algebraic Methods Program. 109 (2019) 100480, https://doi.org/10.1016/j.jlamp.2019.100480, Available at http://arxiv.org/abs/1912.05616.
[27] R.J. van Glabbeek, Reactive bisimulation semantics for a process algebra with time-outs, in: I. Konnov, L. Kovács (Eds.), Proceedings 31st International Conference on Concurrency Theory (CONCUR 20), Online, September 2020, in: Leibniz International Proceedings in Informatics (LIPIcs), vol. 171, Schloss Dagstuhl—Leibniz-Zentrum für Informatik, 2020.
[28] R.J. van Glabbeek, Reactive temporal logic, in: O. Dardha, J. Rot (Eds.), Proc. Combined 27th International Workshop on Expressiveness in Concurrency and 17th Workshop on Structural Operational Semantics, Online, in: Electronic Proceedings in Theoretical Computer Science, vol. 322, Open Publishing Association, 2020, pp. 51–68.
[29] R.J. van Glabbeek, Failure trace semantics for a process algebra with time-outs, Log. Methods Comput. Sci. 17 (2) (2021) 11, https://doi.org/10.23638/LMCS-17(2:11)2021, Available at http://arxiv.org/abs/2002.10814.
[30] R.J. van Glabbeek, U. Goltz, J.-W. Schicke, Abstract processes of place/transition systems, Inf. Process. Lett. 111 (13) (2011) 626–633, https://doi.org/10.1016/j.ipl.2011.03.013, Available at http://arxiv.org/abs/1103.5916.
[31] R.J. van Glabbeek, P. Höfner, Progress, Fairness and Justness in Process Algebra, Technical Report 8501, NICTA, Sydney, Australia, 2015, Available at http://arxiv.org/abs/1501.03268.
[32] R.J. van Glabbeek, P. Höfner, CCS: it's not fair! - fair schedulers cannot be implemented in CCS-like languages even under progress and certain fairness assumptions, Acta Inform. 52 (2–3) (2015) 175–205, https://doi.org/10.1007/s00236-015-0221-6, Available at http://arxiv.org/abs/1505.05964.
[33] R.J. van Glabbeek, P. Höfner, Progress, justness and fairness, ACM Comput. Surv. 52 (4) (2019) 69, https://doi.org/10.1145/3329125, Available at https://arxiv.org/abs/1810.07414.
[34] R.J. van Glabbeek, F.W. Vaandrager, Petri net models for algebraic theories of concurrency (extended abstract), in: J.W. de Bakker, A.J. Nijman, P.C. Treleaven (Eds.), Proc. PARLE, Parallel Architectures and Languages Europe, Eindhoven, the Netherlands, vol. II: Parallel Languages, in: LNCS, vol. 259, Springer, 1987, pp. 224–242.
[35] U. Goltz, A. Mycroft, On the relationship of CCS and Petri nets, in: J. Paredaens (Ed.), Proc. 11th ICALP, Antwerpen, in: LNCS, vol. 172, Springer, 1984, pp. 196–208.
[36] J.F. Groote, M.R. Mousavi, Modeling and Analysis of Communicating Systems, MIT Press, 2014.
[37] C.A.R. Hoare, Communicating Sequential Processes, Prentice Hall, Englewood Cliffs, 1985.
[38] M. Huth, M.D. Ryan, Logic in Computer Science — Modelling and Reasoning About Systems, 2nd edition, Cambridge University Press, 2004.

[39] E. Kindler, R. Walter, Mutex needs fairness, Inf. Process. Lett. 62 (1) (1997) 31–39, https://doi.org/10.1016/S0020-0190(97)00033-1.
[40] L. Kleinrock, Analysis of a time-shared processor, Nav. Res. Logist. Q. 11 (1) (1964) 59–73, https://doi.org/10.1002/nav.3800110105.
[41] D.E. Knuth, Additional comments on a problem in concurrent programming control, Commun. ACM 9 (5) (1966) 321–322, https://doi.org/10.1145/355592.365595.
[42] L. Lamport, A new solution of Dijkstra's concurrent programming problem, Commun. ACM 17 (8) (1974) 453–455, https://doi.org/10.1145/361082.361093.
[43] L. Lamport, Proving the correctness of multiprocess programs, IEEE Trans. Softw. Eng. 3 (2) (1977) 125–143, https://doi.org/10.1109/TSE.1977.229904.
[44] L. Lamport, What good is temporal logic?, in: R.E. Mason (Ed.), Information Processing 83, North-Holland, 1983, pp. 657–668.
[45] L. Lamport, On interprocess communication. Part II: Algorithms, Distrib. Comput. 1 (2) (1986) 86–101, https://doi.org/10.1007/BF01786228.
[46] O. Lichtenstein, A. Pnueli, L.D. Zuck, The glory of the past, in: R. Parikh (Ed.), Proc. Workshop on Logics of Programs, Brooklyn College, New York, NY, USA, in: LNCS, vol. 193, Springer, 1985, pp. 196–218.
[47] R. Milner, Operational and algebraic semantics of concurrent processes, in: J. van Leeuwen (Ed.), Handbook of Theoretical Computer Science, chapter 19, Elsevier Science Publishers B.V. (North-Holland), 1990, pp. 1201–1242, Alternatively see Communication and Concurrency, Prentice-Hall, Englewood Cliffs, 1989, of which an earlier version appeared as a Calculus of Communicating Systems, LNCS 92, Springer, 1980.
[48] J. Nagle, On Packet Switches with Infinite Storage, RFC 970, Network Working Group, Available at http://tools.ietf.org/rfc/rfc970.txt, 1985.
[49] J. Nagle, On packet switches with infinite storage, IEEE Trans. Commun. 35 (4) (1987) 435–438, https://doi.org/10.1109/TCOM.1987.1096782.
[50] M. Nielsen, G.D. Plotkin, G. Winskel, Petri nets, event structures and domains, part I, Theor. Comput. Sci. 13 (1) (1981) 85–108, https://doi.org/10.1016/0304-3975(81)90112-2.
[51] E.-R. Olderog, Operational Petri net semantics for CCSP, in: G. Rozenberg (Ed.), Advances in Petri Nets 1987, in: LNCS, vol. 266, Springer, 1987, pp. 196–223.
[52] G.L. Peterson, Myths about the mutual exclusion problem, Inf. Process. Lett. 12 (3) (1981) 115–116, https://doi.org/10.1016/0020-0190(81)90106-X.
[53] Amir Pnueli, The temporal logic of programs, in: Foundations of Computer Science, FOCS'77, IEEE, 1977, pp. 46–57.
[54] M. Raynal, Concurrent Programming - Algorithms, Principles, and Foundations, Springer, 2013.
[55] A. Silberschatz, P.B. Galvin, G. Gagne, Operating System Concepts, 9th edition, Wiley, 2012, Available at http://os-book.com/OS9/index.html.
[56] M.S.C. Spronck, Safe registers and Aravind's BLRU algorithm for mutual exclusion in mCRL2, Technical Report, Eindhoven University of Technology, 2021, Available at https://www.win.tue.nl/~luttik/BRP/Myrthe_Spronck.pdf.
[57] M.S.C. Spronck & B. Luttik: Process-Algebraic Models of Multi-Writer Multi-Reader Non-Atomic Registers, Personal Communication, (2021).
[58] B.K. Szymański, A simple solution to Lamport's concurrent programming problem with linear wait, in: J. Lenfant (Ed.), Proc. 2nd International Conference on Supercomputing, ICS'88, Saint Malo, France, ACM, 1988, pp. 621–626.
[59] A. Valmari, M. Setälä, Visual verification of safety and liveness, in: M.-C. Gaudel, J. Woodcock (Eds.), Industrial Benefit and Advances in Formal Methods, FME'96, in: LNCS, vol. 1051, Springer, 1996, pp. 228–247.
[60] W. Vogler, Efficiency of asynchronous systems, read arcs, and the MUTEX-problem, Theor. Comput. Sci. 275 (1–2) (2002) 589–631, https://doi.org/10.1016/S0304-3975(01)00300-0.
[61] D.J. Walker, Automated analysis of mutual exclusion algorithms using CCS, Form. Asp. Comput. 1 (1) (1989) 273–292, https://doi.org/10.1007/BF01887209.
[62] G. Winskel, Event structures, in: W. Brauer, W. Reisig, G. Rozenberg (Eds.), Petri Nets: Applications and Relationships to Other Models of Concurrency, Advances in Petri Nets 1986, Part II; Proceedings of an Advanced Course, Bad Honnef, September 1986, in: LNCS, vol. 255, Springer, 1987, pp. 325–392.