Just Verification of Mutual Exclusion Algorithms

Rob van Glabbeek
School of Informatics, University of Edinburgh, UK
School of Computer Science and Engineering, University of New South Wales, Sydney, Australia

Bas Luttik
Eindhoven University of Technology, The Netherlands

Myrthe S.C. Spronck *
Eindhoven University of Technology, The Netherlands

Abstract

We verify the correctness of a variety of mutual exclusion algorithms through model checking. We look at algorithms where communication is via shared read/write registers, where those registers can be atomic or non-atomic. For the verification of liveness properties, it is necessary to assume a completeness criterion to eliminate spurious counterexamples. We use justness as completeness criterion. Justness depends on a concurrency relation; we consider several such relations, modelling different assumptions on the working of the shared registers. We present executions demonstrating the violation of correctness properties by several algorithms, and in some cases suggest improvements.

2012 ACM Subject Classification Theory of computation → Distributed algorithms; Theory of computation → Concurrency; Theory of computation → Verification by model checking

Keywords and phrases Mutual exclusion, safe registers, regular registers, overlapping reads and writes, atomicity, safety, liveness, starvation freedom, justness, model checking, mCRL2.

Related Version An abbreviated version of this paper will appear in Proc. CONCUR'25, Leibniz International Proceedings in Informatics (LIPIcs), Schloss Dagstuhl—Leibniz-Zentrum für Informatik.

Supplementary Material The mCRL2 models and modal μ-calculus formulae can be found in the mCRL2 GitHub repository (commit ff6122b).

Funding *Rob van Glabbeek*: Supported by Royal Society Wolfson Fellowship RSWF\R1\221008

1 Introduction

The mutual exclusion problem is a fundamental problem in concurrent programming. Given $N \geq 2$ *threads*,[1] each of which may occasionally wish to access a *critical section*, a *mutual exclusion algorithm* seeks to ensure that at most one thread accesses its critical section at any given time. Ideally, this is done in such a way that whenever a thread wishes to access its critical section, it eventually succeeds in doing so. Many mutual exclusion algorithms have been proposed in the literature, and in general their correctness depends on assumptions one can make on the environment in which these algorithms will be running. The present paper aims to make these assumptions explicit, and to verify the correctness of some of the most popular mutual exclusion algorithms as a function of these assumptions.

Correctness properties of mutual exclusion algorithms. A thread that does not seek to execute its critical section is said to be executing its *non-critical section*. We regard *leaving the non-critical section* as getting the desire to enter the critical section. After this happens, the thread is executing its *entry protocol*, the part of the mutual exclusion algorithm in

* Corresponding author
[1] What we call threads are in the literature frequently referred to as *processes* or *computers*. We use *threads* to distinguish between the real systems and our models of them, expressed in a process algebra.

which it negotiates with other threads who gets to enter the critical section first. The critical section occurs right after the entry protocol, and is followed by an *exit protocol*, after which the thread returns to its non-critical section. When in its non-critical section, a thread is not expected to communicate with the other threads in any way. Moreover, a thread may choose to remain in its non-critical section forever after. However, once a thread gains access to its critical section, it must leave it within a finite time, so as to make space for other threads.

The most crucial correctness property of a mutual exclusion algorithm is *mutual exclusion*: at any given time, at most one thread will be in its critical section. This is a safety property. In addition, a hierarchy of liveness properties have been considered. The weakest one is *deadlock freedom*: Whenever at least one thread is running its entry protocol, eventually some thread will enter its critical section. This need not be one of the threads that was observed to be in its entry protocol. A stronger property is *starvation freedom*: whenever a thread leaves its non-critical section, it will eventually enter its critical section. A yet stronger property, called *bounded bypass*, augments starvation freedom with a bound on the number of times other threads can gain access to the critical section before any given thread in its entry protocol.

In this paper we check for over a dozen mutual exclusion protocols, and for six possible assumptions on the environment in which they are running, whether they satisfy mutual exclusion, deadlock freedom and starvation freedom. We will not investigate bounded bypass, nor other desirable properties of mutual exclusion protocols, such as *first-come-first-served*, *shutdown safety*, *abortion safety*, *fail safety* and *self-stabilisation* [39].

Memory models.[2] In the mutual exclusion algorithms considered here, the threads communicate with each other solely by reading from and writing to shared registers. The main assumptions on the environment in which mutual exclusion algorithms will be running concern these registers. It is frequently assumed that (read and write) operations on registers are "undividable", meaning that they cannot overlap or interleave each other: if two threads attempt to perform an operation on the same register at the same time, one operation will be performed before the other. This assumption, sometimes referred to as *atomicity*, is explicitly made in Dijkstra's first paper on mutual exclusion [21]. Atomicity is sometimes conceptualised as operations occurring at a single moment in time. We instead acknowledge that operations have duration. Consequently, if operations cannot overlap in time, then, when multiple operations are attempted simultaneously, the one performed first must postpone the occurrence of the others by at least its own duration. One operation postponing another is called *blocking* [19].

Deviating from Dijkstra's original presentation, several authors have considered a variation of the mutual exclusion problem where the atomicity assumption is dropped [37, 48, 38, 39, 53, 54, 4, 6]. Attempted operations can then occur immediately, without blocking each other. We say these operations are *non-blocking*. In this context, read and write operations may be *concurrent*, i.e. overlap in time. We must then consider the consequences of operations overlapping each other.

In [40, 41], Lamport proposes a hierarchy of three memory models in this context,

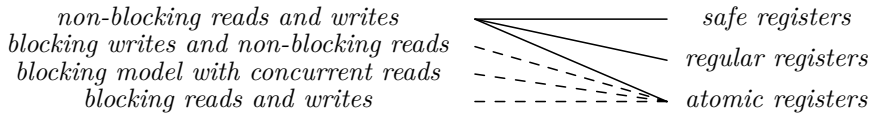
[2] A *memory model* describes the interactions of threads through memory and their shared use of the data. The models reviewed here differ in the degree in which different register accesses exclude each other, and in what values a register may return in case of overlapping reads and writes. In this paper, we do not consider *weak memory models*, that allow for compiler optimisations, and for reads to sometimes fetch values that were already changed by another thread. In [8] is has been shown that mutual exclusion cannot be realised in weak memory models, unless those models come with *memory fences* or *barriers* that can be used to undermine their weak nature.

specifically for single-writer multi-reader (SWMR) registers; such registers are owned by one thread, and only that thread is capable of writing to it. Crucial for these definitions is the assumption that every register has a domain, and a read of that register always yields a value from that domain. It is also important to note that threads can only perform a single operation at a time, meaning that a thread's operations can never overlap each other.

- A **safe** register guarantees merely that a read that is not concurrent with any write returns the most recently written value.
- A **regular** register guarantees that any read returns either the last value written before it started, or the value of any overlapping write, if there is one.
- An **atomic** register guarantees that reads and writes behave as though they occur in some total order. This total order must comply with the real-time ordering of the operations: if operation a ends before operation b begins, then a must be ordered before b.

In Appendix A we illustrate the differences between these memory models with an example. They form a hierarchy, in the sense that any atomic register is regular, and any regular one is safe. When we merely know that a register is safe, a read that overlaps with any write might return any value in the domain of the register. In Section 3 we discuss the generalisation of these memory models to multi-writer multi-reader (MWMR) registers, ones that can be written and read by all threads.

Besides blocking and non-blocking registers, as explained above, we consider two intermediate memory models. The *blocking model with concurrent reads* requires (1) any scheduled read or write to await the completion of any write that is in progress, and (2) any scheduled write to await the completion of any unfinished read. However, reads from different threads need not wait for each other and may overlap in time without ill effects. In the model of *non-blocking reads*,[3] we have (1) but not (2). This model, where writes block reads but reads do not block writes, may apply when writes can abort in-progress reads, superseding them.



In this paper, we model six different memory models, which are illustrated above. The blocking aspect of our memory models is captured via different concurrency relations (Section 5). The distinction between safe, regular and atomic registers is captured via three different process algebraic models (Section 3). Since the safe/regular/atomic distinction is only relevant in models that allow writes to overlap reads and writes, we only make it for the non-blocking model; for the other three memory models we reuse our atomic register models.

Completeness criteria. In previous work [51], we checked the mutual exclusion property of several algorithms, with safe, regular and atomic MWMR registers, through model checking with the mCRL2 toolset [17]. We did not check the liveness properties at that time; the presence of certain infinite loops in our models introduced spurious counterexamples to such properties, which hindered our verification efforts. As an example of what we call a "spurious counterexample", we frequently found violations to starvation freedom where one thread, i, never obtained access to its critical section because a different thread, j, was endlessly repeating a busy wait, or some other infinite cycle which should reasonably not prevent i from progressing to its critical section. Yet, the model checker does not know this, and can therefore only conclude that the property is not satisfied. In this paper, we extend our

[3] In this terminology, from [19], a *blocking read* blocks a write; it does not refer to a read that is blocked.

previous work by addressing this problem and checking liveness properties as well.

One method for discarding spurious counterexamples from verification results is applying completeness criteria: rules for determining which paths in the model represent real executions of the modelled system. By ensuring that all spurious paths are classified as incomplete and only taking complete paths into consideration when verifying liveness properties, we can circumvent the spurious counterexamples. Of course, one must take care not to discard true system executions by classifying those as incomplete. The completeness criterion must therefore be chosen with care. Examples of well-known completeness criteria are weak fairness and strong fairness. Weak fairness assumes that every task[4] that eventually is perpetually enabled must occur infinitely often; strong fairness assumes that if a task is infinitely often enabled it must occur infinitely often [43, 5, 29]. In effect, making a fairness assumptions amounts to assuming that if something is tried often enough, it will always eventually succeed [29]. In that sense, these assumptions, even weak fairness, are rather strong, and may well result in true system executions being classified as incomplete. In this paper, we therefore use the weaker completeness criterion *justness* [29, 26, 14].

Unlike weak and strong fairness, justness takes into account how different actions in the model relate to each other. Informally, it says that if an action a can occur, then eventually a occurs itself, or a different action occurs that interferes with the occurrence of a. The underlying idea of justness is that the different components that make up a system must all be capable of making progress: if thread i wants to perform an action entirely independent of the actions performed by j, then there can be no interference. However, if both threads are interacting with a shared register, then we may decide that one thread writing to the register can prevent the other from reading it at the same time, or vice versa. Which actions interfere with each other is a modelling decision, dependent on our understanding of the real underlying system. It is formalised through a *concurrency relation*, which must adhere to some restrictions. In this paper we propose four concurrency relations, each modelling one of the four major memory models reviewed above: non-blocking reads and writes, blocking writes and non-blocking reads, the blocking model with concurrent reads, and blocking reads and writes.

Model checking. Traditionally, mutual exclusion algorithms have been verified by pen-and-paper proofs using behavioural reasoning. As remarked by Lamport [39], "the behavioral reasoning used in our correctness proofs, and in most other published correctness proofs of concurrent algorithms, is inherently unreliable". This is especially the case when dealing with the intricacies of non-atomic registers.[5] This problem can be alleviated by automated formal verification; here we employ model checking.

While the precise modelling of the algorithms, the registers and the employed completeness criterion requires great care, the subsequent verification requires a mere button-push and some patience. Since our model checker traverses the entire state-space of a protocol, the verified protocols and all their registers need to be finite. This prevented us from checking the bakery algorithm [37], as it is one of the few mutual exclusion protocols that employs an unbounded state space. Moreover, those algorithms that work for N threads, for any $N \in \mathbb{N}$,

[4] What constitutes a *task* differs from paper to paper; hence there are multiple flavours of strong and weak fairness; here a task could be a read or write action of a certain thread on a certain register.

[5] A good illustration of unreliable behavioural reasoning is given in [27, Section 21], through a short but fallacious argument that the mutual exclusion property of Peterson's mutual exclusion protocol, which is known to hold for atomic registers, would also hold for safe registers. We challenge the reader to find the fallacy in this argument before looking at the solution.

could be checked for small values of N only; in this paper we take $N = 3$. Consequently, any failure of a correctness property that shows up only for > 3 threads will not be caught here.

As stated, we employed these methods in previous work to check mutual exclusion algorithms. Although there we checked only safety properties, and did not consider the blocking aspects of memory, this already gave interesting results. For instance, we showed that Szymanski's flag algorithm from [53], even when adapted to use Booleans, violates mutual exclusion with non-atomic registers. Here, we expand this previous work by checking deadlock freedom and starvation freedom in addition to mutual exclusion, and by including blocking into our memory models. In total, we check the three correctness properties of over a dozen mutual exclusion algorithms, for six different memory models. Among others, we cover Aravind's BLRU algorithm [6], Dekker's algorithm [20, 3] and its RW-safe variant [16], and Szymanski's 3-bit linear wait algorithm [54]. In some cases where we find property violations, we suggest fixes to the algorithms so that the properties are satisfied.

2 Preliminaries

A *labelled transition system* (LTS) is a tuple $(\mathcal{S}, Act, init, Trans)$ in which $\mathcal{S}$ is a finite set of states, Act is a finite set of actions, $init \in \mathcal{S}$ is the initial state, and $Trans \subseteq \mathcal{S} \times Act \times \mathcal{S}$ is a transition relation. We write $s \xrightarrow{a} s'$ for $(s, a, s') \in Trans$. We say an action a is *enabled* in a state s if there exists a state s' such that $(s, a, s') \in Trans$.

A *path* π is a non-empty, potentially infinite alternating sequence of states and actions $s_0 a_1 s_1 a_2 \ldots$, with $s_0, s_1, \ldots \in \mathcal{S}$ and $a_1, a_2, \ldots \in Act$, such that if π is finite, then its last element is a state, and for all $i \in \mathbb{N}$, $s_i \xrightarrow{a_{i+1}} s_{i+1}$. The first state of π is its *initial state*. The *length* of π is the number of transitions in it.

We use a notion of parallel composition that is taken from Hoare's CSP [34], where synchronisation between components is enforced on all shared actions. It is defined as follows: For some $k \geq 1$, let $P_1, \ldots, P_k$ be LTSs, where $P_i = (\mathcal{S}_i, Act_i, init_i, Trans_i)$ for all $1 \leq i \leq k$. The *parallel composition* $P_1 \parallel \ldots \parallel P_k$ of $P_1, \ldots, P_k$ is the LTS $P = (\mathcal{S}, Act, init, Trans)$ in which $\mathcal{S} = \mathcal{S}_1 \times \ldots \times \mathcal{S}_k$, $Act = \bigcup_{1 \leq i \leq k} Act_i$, $init = (init_1, \ldots, init_k)$, and a transition $((s_1, \ldots, s_k), a, (s'_1, \ldots, s'_k))$ is in $Trans$ if, and only if, $a \in Act$ and the following are true for all $1 \leq i \leq k$: if $a \notin Act_i$, then $s_i = s'_i$, and if $a \in Act_i$, then $(s_i, a, s'_i) \in Trans_i$.
Note that by this definition of $Trans$, if an action is in the action set of a component but not enabled by that component in a particular state of the parallel composition, then the composition cannot perform a transition labelled with that action.

As mentioned in the introduction, the completeness criterion we use for our liveness verification is a variant of *justness* [26, 29]. Specifically, while justness is originally defined on transitions, we here define it on action labels, an adaption we take from [14]. As stated earlier, the definition of justness relies on the notion of a concurrency relation.

▶ **Definition 1.** Given an LTS $(\mathcal{S}, Act, init, Trans)$, a relation $\smile\!\!\!\bullet \subseteq Act \times Act$ is a *concurrency relation* if, and only if:

- $\smile\!\!\!\bullet$ is irreflexive.
- For all $a \in Act$, if π is a path from a state $s \in \mathcal{S}$ to a state $s' \in \mathcal{S}$ such that a is enabled in s and $a \smile\!\!\!\bullet b$ for all $b \in Act$ occurring on π, then a is enabled in s'.

A concurrency relation may be asymmetric. We often reason about the complement of $\smile\!\!\!\bullet$, $\not\smile\!\!\!\bullet$. Read $a \smile\!\!\!\bullet b$ as "a is independent from b" and $a \not\smile\!\!\!\bullet b$ as "b interferes with/postpones a".

▶ **Observation 2.** Concurrency relations can be refined by removing pairs; a subset of a concurrency relation is still a concurrency relation.

Informally, justness says that a path is complete if whenever an action a is enabled along the path, there is eventually an occurrence of an action (possibly a itself) that interferes with it. This can be weakened by defining a set of *blockable actions*, for which this restriction does not hold; a blockable action may be enabled on a complete path without there being a subsequent occurrence of an interfering action. In this paper, the action of a thread to leave its non-critical section will be blockable. This way we model that a thread may choose to never take that option. We give the formal definition of justness, incorporating the blockable actions. We represent the set of blockable actions as $\mathcal{B}$. Its complement, $\overline{\mathcal{B}}$, is defined as $Act \setminus \mathcal{B}$, given a set of actions Act.

▶ **Definition 3.** A path π in an LTS $(\mathcal{S}, Act, init, Trans)$ satisfies $\mathcal{B}$-$\smile\!\!\!\bullet$-*justness of actions* ($JA^{\smile\bullet}_{\mathcal{B}}$) if, and only if, for each suffix π' of π, if an action $a \in \overline{\mathcal{B}}$ is enabled in the initial state of π', then an action $b \in Act$ occurs in π' such that $a \not\smile\!\!\!\bullet\ b$.

We say that a property is satisfied on a model under $JA^{\smile\bullet}_{\mathcal{B}}$ if it is satisfied on every path of that model, starting from the model's initial state, that satisfies $JA^{\smile\bullet}_{\mathcal{B}}$. If $\mathcal{B}$ and $\smile\!\!\!\bullet$ are clear from the context, we simply say that a path that satisfies $JA^{\smile\bullet}_{\mathcal{B}}$ is *just*.

3 Register models

In [51], we presented process-algebraic models of MWMR safe, regular and atomic registers. Through the semantics of the process algebra, this determines an LTS for each register of a given kind. In this paper, we use the same definitions for the three register types, but we have altered the process-algebraic models to be more compact and better facilitate the definition of the concurrency relations. The process-algebraic models can be found in Appendix B; here we merely summarise the key design decisions.

A register model represents a multi-reader multi-writer read-write register that allows every thread to read from and write to it. However, every thread may only perform a single operation on the register at a time. The register model specifies the behaviour of the register in response to operations performed by threads. Here we presuppose two disjoint finite sets: $\mathbb{T}$ of thread identifiers (thread id's) and $\mathbb{R}$ of register identifiers (register id's). Additionally, for every $r \in \mathbb{R}$ we reference the set $\mathbb{D}_r$ of all data values that the register r can hold.

Recall that read and write operations take time, and may hence be concurrent. Therefore we represent a single operation with two actions: an *invocation* to indicate the start of the operation, and a *response* to indicate its end. Two operations are concurrent if the interval between their respective invocations and responses overlaps. The interface of a register is represented by the following actions: a read by thread $t \in \mathbb{T}$ of register $r \in \mathbb{R}$ that returns value $d \in \mathbb{D}_r$ is a start read action $sr_{t,r}$ followed by a finish read action $fr_{t,r}(d)$; a write of value $d \in \mathbb{D}_r$ by a thread $t \in \mathbb{T}$ to register $r \in \mathbb{R}$ is a start write action $sw_{t,r}(d)$ followed by a finish write action $fw_{t,r}$. In addition to this interface, which the threads can use to perform operations on the register, the regular and atomic models also use *register local actions*; these are internal actions by the register that are used to model the correct behaviour.

A register model requires some finite amount of memory to store a representation of relevant past events. We store this in what we call the *status object*, which features a finite set $\mathbb{S}$ of possible states. We abstract away from the exact implementation; for this presentation, all that is relevant is which information can be retrieved from it. Amongst others, we use the following *access functions*, which are local to any given register r:

- $stor : \mathbb{S} \to \mathbb{D}_r$, the value that is currently stored in the register.
- $wrts : \mathbb{S} \to 2^{\mathbb{T}}$, the set of thread id's of threads that have invoked a write operation on this register that has not yet had its response.

Any occurrence of a register action a induces a state change $s \xrightarrow{a} s'$, resulting in an *update* to these access functions. For instance, the actions $sw_{t,r}(d)$ and $fw_{t,r}$ cause the updates $wrts(s') = wrts(s) \cup \{t\}$ and $wrts(s') = wrts(s) \setminus \{t\}$, respectively.

3.1 Safe MWMR registers

To extend the single-writer definition of safe registers to a multi-writer one, we follow Lamport in assuming that a concurrent read cannot affect the behaviour of a read or write. Lamport's SWMR definitions consider how concurrent writes affect reads, but not how concurrent writes affect writes. Here we follow Raynal's approach for safe registers: a write that is concurrent with another write sets the value of the register to some arbitrary value in the domain of the register [49]. We can summarise the behaviour of MWMR safe registers with four rules:

1. A read not concurrent with any writes on the same register returns the value most recently written into the register.
2. A read concurrent with one or more writes on the same register returns an arbitrary value in the domain of the register.
3. A write not concurrent with any other write on the same register results in the intended value being set.
4. A write concurrent with one or more other writes on the same register results in an arbitrary value in the domain of the register being set.

In our model of a safe register r, its status object maintains a Boolean variable *ovrl* for each thread id, telling whether an ongoing read or write action of this thread overlapped with a write by another thread. The value of *ovrl* is updated in a straightforward way each time r experiences a register interface action $sr_{t,r}$, $fr_{t,r}(d)$, $sw_{t,r}(d)$ or $fw_{t,r}$, aided by the access function *wrts*. Using this function, our model can determine which of the above four rules applies when a read or write finishes, and behave accordingly.

3.2 Regular MWMR registers

We wish to define regular MWMR registers as an extension of Lamport's definition of SWMR regular registers: a read returns either the last written value before the read began, or the value of any concurrent write, if there is one. This is non-trivial; in [51] we present one extension and compare it to four different suggestions from [50]. The complexity comes from determining what the last written value is, given that writes may be concurrent with each other. Here, following [51], we require that all threads see the same global ordering on writes once those writes have completed. Hence, if two writes w_1 and w_2 occur concurrently, and after their completion, but before the invocation of any other write, there are two reads r_1 and r_2, then either both r_1 and r_2 see w_1's value as the last written value, or they both see w_2's value as the last written value. We generate the global ordering through the register local order write action $ow_{t,r}$, which is scheduled between the start write action $sw_{t,r}(d)$ and the finish write $fw_{t,r}$. This action does not represent any true internal behaviour by the register; the interleaving of order write actions from various threads merely determines the global ordering. Given a read, we say the "last written" value this read sees, and hence the value this read may return in addition to those of overlapping writes, is the intended value of the write whose ordering was the most recent before the read's invocation.

In our process algebraic model, the value *stor* of the register is set when an action $ow_{t,r}$ occurs. During any read action of a thread t, that is, between $sr_{t,r}$ and $fr_{t,r}(d)$, the register model builds a set of the possible return values on the fly. When the read starts, this set is initialised to *stor* and the intended value of every active write. Subsequently, whenever a

write occurs, its intended value is added to the set. This way, at the finish read, the set will contain exactly those values that the read could return.

3.3 Atomic MWMR registers

Lamport's definition of SWMR atomic registers, namely that the register must behave as though reads and writes occur in some strict order, is directly applicable to the MWMR case. We reuse the register local order write action from the regular register model, and add the similar order read action $or_{t,r}$ for read operations. This way, we generate an ordering on all operations. In our process-algebraic model, *stor* is updated when the $ow_{t,r}$ occurs, similar to regular registers. For read operations, the value of *stor* when $or_{t,r}$ occurs is remembered, and returned at the matching response.

4 Thread-register models

In our models, we combine processes representing both threads and registers. Similar to how register processes may contain register local actions, threads may contain *thread local actions*. Crucially, the local actions of both registers and threads are not involved in any communication, meaning that the only way for two threads to communicate is by writing to and reading from registers using the register interface actions.

The register models are mostly independent of the algorithm that we analyse with them; the algorithm merely dictates a register's identifier, domain, and initial value. However, the thread models are fully dependent on the modelled algorithm, which dictates their behaviour. Therefore, we cannot present an algorithm-independent thread process. Instead, we presuppose the existence of an LTS $T_t = (\mathcal{S}_t, Act_t, init_t, Trans_t)$ and a set of thread local actions $TLoc_t$ for every $t \in \mathbb{T}$ such that $Act_t = \{sr_{t,r}, fr_{t,r}(d), sw_{t,r}(d), fw_{t,r} \mid r \in \mathbb{R}, d \in \mathbb{D}_r\} \cup TLoc_t$. We assume that all sets of thread local actions are pairwise disjoint and that all thread LTSs T_t satisfy two properties, representing the reasonable implementation of read and write operations. Firstly, on all paths from $init_t$, each transition labelled $sr_{t,r}$ for some $r \in \mathbb{R}$ must go to a state where exactly the actions $fr_{t,r}(d)$ for all $d \in \mathbb{D}_r$ are enabled. Similarly, all transitions labelled $sw_{t,r}(d)$ for some $r \in \mathbb{R}, d \in \mathbb{D}_r$ must go to states where only $fw_{t,r}$ is enabled. Secondly, transitions labelled $fr_{t,r}(d)$ or $fw_{t,r}$ are only enabled in these states.

We combine thread and register LTSs into *thread-register models*. For the following definition, we let $R_r = (\mathcal{S}_r, Act_r, init_r, Trans_r)$ be the LTS associated with each $r \in \mathbb{R}$.

▶ **Definition 4.** A *thread-register model* is a six-tuple $(\mathcal{S}, Act, init, Trans, thr, reg)$, such that $(\mathcal{S}, Act, init, Trans)$ is a parallel composition of thread and register LTSs and

- $thr\colon Act \to \mathbb{T}$ is a mapping from actions to thread id's; and
- $reg\colon Act \to \mathbb{R} \cup \{\bot\}$ is a mapping from actions to register id's and the special value $\bot \notin \mathbb{R}$.

thr and *reg* are defined in the obvious way, e.g., $thr(sr_{t,r}) = t$ and $reg(sr_{t,r}) = r$. Crucially, for a thread local action a, $reg(a) = \bot$.

Note that by our construction of the thread and register LTSs, every action in Act appears in at most two components of the parallel composition. Specifically, for all $t \in \mathbb{T}$ and $a_t \in TLoc_t$, a_t appears only in T_t; for all $t \in \mathbb{T}, r \in \mathbb{R}$, $or_{t,r}$ and $ow_{t,r}$ appear only in R_r; and for all $t \in \mathbb{T}, r \in \mathbb{R}$ and $d \in \mathbb{D}_r$, $sr_{t,r}, fr_{t,r}(d), sw_{t,r}(d)$ and $fw_{t,r}$ appear exactly in T_t and R_r.

5 Justness for thread-register models

In order to obtain a suitable notion of justness for our thread-register models, we need to choose both $\mathcal{B}$ and $\smile\!\!\bullet$. Only thread local actions will be blockable; we define $\mathcal{B}$ in Section 6.

The concurrency relation, on the other hand, should relate the register interface actions. This is how we represent whether it is reasonable for one thread's operations on a register to interfere with (and thereby postpone) another thread's operations on that register. We use four different concurrency relations in our verifications, representing the four different models of blocking described in Section 1. These concurrency relations do not reference the thread local actions outside of the *thr* mapping, so we can already present these relations before giving more details on the precise models. To establish that the relations we present are indeed concurrency relations, we first establish a property of our models. We call this property *thread consistency*.

▶ **Definition 5.** An LTS $(\mathcal{S}, Act, init, Trans)$ is *thread consistent* with respect to a mapping $thr : Act \to \mathbb{T}$ if, and only if, for all states $s \in \mathcal{S}$, if an action $a \in Act$ is enabled in s and there exists a transition $s \xrightarrow{b} s'$ for some $s' \in \mathcal{S}, b \in Act$ such that $thr(a) \neq thr(b)$, then a is also enabled in s'.

The correctness of our concurrency relations (cf. Definition 1) relies on our thread-register models being thread-consistent. The proof of this fact is given in Appendix C.

▶ **Lemma 6.** Let $M = (\mathcal{S}, Act, init, Trans, thr, reg)$ be a thread-register model. Then the LTS $(\mathcal{S}, Act, init, Trans)$ is thread consistent with respect to the mapping thr.

For the definitions of our four concurrency relations, we fix a thread-register model $M = (\mathcal{S}, Act, init, Trans, thr, reg)$. We also introduce two predicates on Act: $sr?$ and $sw?$. For an action $a \in Act$, these are defined as:

$$sr?(a) = \exists_{t\in\mathbb{T}, r\in\mathbb{R}}.(a = sr_{t,r}) \qquad sw?(a) = \exists_{t\in\mathbb{T}, t\in\mathbb{R}, d\in\mathbb{D}_r}.(a = sw_{t,r}(d)).$$

The *thread interference relation*, $\smile\!\!\bullet_T$, expresses that every action is independent from every other action *unless* the two actions belong to the same thread; every two actions by the same thread interfere with each other. It captures the memory model with non-blocking reads and writes. This is the coarsest concurrency relation we will use.

▶ **Definition 7.** $\smile\!\!\bullet_T = \{(a, b) \mid a, b \in Act, thr(a) \neq thr(b)\}$

▶ **Lemma 8.** $\smile\!\!\bullet_T$ is a concurrency relation for M.

This follows by a straightforward application of Lemma 6. The details are in Appendix D.

The model with blocking writes and non-blocking reads is captured by the *signalling reads relation*, $\smile\!\!\bullet_S$.

▶ **Definition 9.** $\smile\!\!\bullet_S = \smile\!\!\bullet_T \setminus \{(a, b) \mid a, b \in Act, sr?(a) \vee sw?(a), sw?(b), reg(a) = reg(b)\}$

Intuitively, this is the same as $\smile\!\!\bullet_T$ except that one thread starting a write to a register can interfere with a write to or a read from that same register by another thread. However, a read cannot interfere with another thread's read or write. This concurrency relation has a precedent in [23, 14]. There, reads are modelled as *signals*, which differ from standard actions in that they do not block any other actions. Hence the name of this relation.

The blocking model with concurrent reads is captured by the *interfering reads relation*, $\smile\!\!\bullet_I$. This is a further refinement from $\smile\!\!\bullet_S$, where a start read can interfere with a start write on the same register, but cannot interfere with a start read.

▶ **Definition 10.** $\smile\!\!\bullet_I = \smile\!\!\bullet_S \setminus \{(a, b) \mid a, b \in Act, sw?(a), sr?(b), reg(a) = reg(b)\}$

This goes with the idea that performing a write on a memory location can only be done when the memory is reserved: repeated reads can prevent the memory from being reserved for a write, but as long as there is no write all the reads can take place concurrently.

Finally, the model of blocking reads and writes is captured by the *all interfering relation*, $\smile\!\!\bullet_A$, a refinement of $\smile\!\!\bullet_I$ where a start read can also interfere with another start read.

▶ **Definition 11.** $\smile\!\!\bullet_A = \smile\!\!\bullet_I \setminus \{(a,b) \mid a,b \in Act, sr?(a), sr?(b), reg(a) = reg(b)\}$

In this model, every operation on a register fully reserves that register for only that operation, and hence can prevent any other operation from taking place at the same time.

▶ **Lemma 12.** $\smile\!\!\bullet_S$, $\smile\!\!\bullet_I$ and $\smile\!\!\bullet_A$ are concurrency relations for M.

Proof. This follows from Lemma 8 and Observation 2. ◀

As stated in the introduction, we capture six memory models. We obtain three variants of the non-blocking model by combining the $\smile\!\!\bullet_T$ relation with the safe, regular and atomic register models. The remaining three memory models are represented by combining $\smile\!\!\bullet_S$, $\smile\!\!\bullet_I$ and $\smile\!\!\bullet_A$ with the atomic register model. In Appendix E, we formally characterise just paths in our thread-register models for all six variants. In Appendix F, we prove that using the atomic register model, which allows overlapping writes, for the three memory models with blocking writes is sound for verification purposes.

6 Verification

Below, and in Appendix I, we collect over a dozen mutual exclusion algorithms from the literature. We also present altered versions of several algorithms, incorporating fixes we propose. All these algorithms, and the registers themselves, have been translated to the process algebra mCRL2 [31]. The mCRL2 files are available as supplementary material. We give the most important design decisions regarding this translation here; further details can be found in Appendix G. The only operations on registers we allow are reading and writing. Hence, more complicated statements that may have been present in the original presentation of an algorithm, such as compare-and-swap instructions, are converted into these primitive operations. A statement like "**await** $\forall_{i \in \mathbb{T}} : x(i)$", where x is some condition on a thread id i, is modelled as a recursive process that checks each thread id from smallest to largest, waiting for each until $x(i)$ is satisfied. Where an algorithm does not specify the initial value of a register, we take the lowest value from the given domain.

We use c_t and nc_t, with $t \in \mathbb{T}$, as thread local actions. These actions represent a thread entering its critical section and leaving its non-critical section, respectively. We define $\mathcal{B} = \{nc_t \mid t \in \mathbb{T}\}$ to capture that a thread may always choose to remain in its non-critical section indefinitely; this is an important assumption underlying the correctness of mutual exclusion protocols.

We did our verification with the mCRL2 toolset [17]. To this end, we encoded mutual exclusion, deadlock freedom, and starvation freedom in the modal μ-calculus, the logic used to represent properties in the mCRL2 toolset. We used the patterns from [52] to incorporate the justness assumption into our formulae for deadlock freedom and starvation freedom. The full modal μ-calculus formulae appear in Appendix H (and also as supplementary material). Besides the correctness properties discussed in Section 1, Dijkstra [21] requires that the correctness of the algorithms may not depend on the relative speeds of the threads. This requirement is automatically satisfied in our approach, since we allow all possible interleavings of thread actions in our models.

We checked mutual exclusion, deadlock freedom, and starvation freedom. If mutual exclusion is not satisfied, we do not care about the other two properties. Additionally, if deadlock freedom is not satisfied, we know that starvation freedom is not satisfied either.

We can therefore summarise our results in a single table: X if none of the three properties are satisfied, M if only mutual exclusion is satisfied, D if only mutual exclusion and deadlock freedom hold, and S if all three are satisfied. See Table 1. As stated previously, we verify liveness properties under justness, where we employ $\smile\!\!\bullet_T$ for safe and regular registers and all four concurrency relations $\smile\!\!\bullet_T$, $\smile\!\!\bullet_S$, $\smile\!\!\bullet_I$ and $\smile\!\!\bullet_A$ for atomic registers. We checked with 2 threads for algorithms designed for 2 threads, and with 3 for all others. We restrict ourselves to at most 3 threads because, due to the state-space explosion problem, even models with only 3 threads frequently take hours or even days to check these properties on.

We list the origin of each algorithm in the table; the results of verifying our proposed alternate versions are indicated by "alt.". We give pseudocode for the algorithms. Therein we merely present the entry and exit protocols of an algorithm, separated by the instruction **critical section**. Implicitly these instructions alternate with the non-critical section, and may be repeated indefinitely. We use N for the number of threads. As identifiers for threads, we use the integers $0 \ldots N{-}1$. So $\mathbb{T} = \{0, \ldots, N{-}1\}$. When presenting pseudocode, we give the algorithm for an arbitrary thread i. When $N = 2$, we use the shorthand notation $j = 1 - i$.

In the subsequent sections, we discuss the most interesting results. Further discussion, and the remaining pseudocode, appears in Appendix I.

6.1 Impossibility of liveness with $\smile\!\!\bullet_I$

Perhaps the most notable result in Table 1 is that no algorithm satisfies either liveness property under $JA^{\smile\!\!\bullet_I}_{\mathcal{B}}$ or $JA^{\smile\!\!\bullet_A}_{\mathcal{B}}$. Since $\smile\!\!\bullet_A$ is a refinement of $\smile\!\!\bullet_I$, we focus on the behaviour for $\smile\!\!\bullet_I$. When we take $\smile\!\!\bullet_I$ as our concurrency relation, then one thread's read of a register can interfere with another thread's write to that same register. It turns out that when this is the case, starvation freedom is impossible for algorithms that rely on communication via registers. The following argument is adapted from [25, 27]. Assume that *Alg* is an algorithm that satisfies starvation freedom. Let i and j be different threads, and assume that all other threads, if any, stay in their non-critical section forever. Since *Alg* is starvation-free, thread i must be capable of freely entering the critical section if thread j is not competing for access. Hence, thread j must communicate its interest in the critical section to thread i as part of its entry protocol. Since reading from and writing to registers is the only form of communication we allow, thread j must, in its entry protocol, write to some register *reg*, which i must read in its own entry protocol. As long as i does not read j's interest from *reg*, thread i can enter the critical section freely. Therefore, if thread i's read of *reg* can block thread j's write to *reg*, thread i can infinitely often access the critical section without ever letting thread j communicate its interest, thus never letting thread j enter.

For this argument it is crucial that right after i's read of *reg*, thread i enters and then leaves the critical section and returns to its entry protocol, where it engages in another read of *reg*, so quickly that thread j has not yet started its write to *reg* in the meantime. This uses the requirement on mutual exclusion protocols that their correctness may not depend on the relative speeds of the threads. Without that requirement one can easily achieve starvation freedom even with blocking reads, as demonstrated in [27].

The argument above explains why starvation freedom is never satisfied for $\smile\!\!\bullet_I$ or $\smile\!\!\bullet_A$. However, it does not explain why we also never observe deadlock freedom. After all, in the execution sketched above, while thread j is stuck in its entry protocol, thread i infinitely often accesses the critical section. While we do not (yet) have an argument that deadlock freedom is impossible to satisfy if reads can block writes for all possible algorithms, we do observe this to be the case for all algorithms we have analysed.

Table 1 Verification results.

Algorithm	*# threads*	*Safe*	*Regular*	*Atomic*			
		T	*T*	*T*	*S*	*I*	*A*
Anderson [4]	2	S	S	S	S	M	M
Aravind BLRU [6]	3	S	S	S	M	M	M
Aravind BLRU (alt.)	3	S	S	S	S	M	M
Attiya-Welch (orig.) [9]	2	D	S	S	D	M	M
Attiya-Welch (orig., alt.)	2	S	S	S	D	M	M
Attiya-Welch (var.) [50]	2	M	M	S	D	M	M
Attiya-Welch (var., alt.)	2	S	S	S	D	M	M
Burns-Lynch [18]	3	D	D	D	D	M	M
Dekker [3]	2	M	M	S	D	M	M
Dekker (alt.)	2	M	M	S	S	M	M
Dekker RW-safe [16]	2	S	S	S	D	M	M
Dekker RW-safe (DFtoSF)	2	S	S	S	S	M	M
Dijkstra [21]	3	M	D	D	M	M	M
Kessels [35]	2	X	X	S	S	M	M
Knuth [36]	3	M	S	S	M	M	M
Lamport 1-bit [39]	3	D	D	D	D	M	M
Lamport 1-bit (DFtoSF)	3	S	S	S	S	M	M
Lamport 3-bit [39]	3	S	S	S	S	M	M
Peterson [47]	2	X	X	S	S	M	M
Szymanski flag (int.) [53]	3	X	X	S	S	M	M
Szymanski flag (bit) [53]	3	X	X	X	X	X	X
Szymanski 3-bit lin. wait [54]	3	X	X	X	X	X	X
Szymanski 3-bit lin. wait (alt.)	2	S	S	S	S	M	M

Algorithm 1 Peterson's algorithm

1: $flag[i] \leftarrow true$
2: $turn \leftarrow i$
3: **await** $flag[j] = false \vee turn = j$
4: **critical section**
5: $flag[i] \leftarrow false$

Algorithm 2 Dekker's algorithm

1: $flag[i] \leftarrow true$
2: **while** $flag[j] = true$ **do**
3: **if** $turn = j$ **then**
4: $flag[i] \leftarrow false$
5: **await** $turn = i$
6: $flag[i] \leftarrow true$
7: **critical section**
8: $turn \leftarrow j$
9: $flag[i] \leftarrow false$

Algorithm 3 Aravind's BLRU algorithm

1: $flag[i] \leftarrow true$
2: **repeat**
3: $stage[i] \leftarrow false$
4: **await** $\forall_{j \neq i} : flag[j] = false \vee date[i] < date[j]$
5: $stage[i] \leftarrow true$
6: **until** $\forall_{j \neq i} : stage[j] = false$
7: **critical section**
8: $date[i] \leftarrow \max(date[0], ..., date[N-1]) + 1$
9: **if** $date[i] \geq 2N - 1$ **then**
10: $\forall_{j \in [0...N-1]} : date[j] \leftarrow j$
11: $stage[i] \leftarrow false$
12: $flag[i] \leftarrow false$

For many algorithms, it is possible for both competing threads to become stuck in their entry protocol. Consider, for example, Peterson's algorithm from [47], here given as Algorithm 1. If *turn* is initially 0, and thread 1 manages to set *flag*[1] to *true* before thread 0 starts the competition, then on line 3 thread 0 will get stuck in a busy waiting loop. Thread 1 needs to set *turn* to 1 to let thread 0 pass line 3, but thread 0's repeated reads of *turn* prevent this write from taking place, resulting in both threads being trapped in the entry protocol. An alternative way to get a deadlock freedom violation is via the exit protocol. Once a thread has finished its critical section access, it needs to communicate that it no longer requires access to the other thread. In Peterson's, this is done on line 5 by setting the thread's *flag* to *false*. However, if the other thread is repeatedly reading this register, such as is done on line 3, then the completion of the exit protocol can be blocked, once again preventing both threads from accessing their critical sections.

We see similar behaviour in all algorithms we analyse. Frequently, although not always, the problem lies in busy waiting loops. Given this behaviour, it would be interesting to modify our models to treat busy waiting reads differently from normal reads, and only allow normal reads to interfere with writes. This would give us greater insight into whether for some of the algorithms it is truly the busy waiting that is the source of the deadlock freedom violation. We leave this as future work.

6.2 Aravind's BLRU algorithm

Aravind's BLRU algorithm [6], here given as Algorithm 3, is designed for an arbitrary number of threads N. Every thread has three registers: *flag* and *stage*, Booleans that are initialised at *false*, and a natural number *date*, initialised at the thread's own id. We observe that this algorithm satisfies all three properties with safe and regular registers, as claimed in [6]. However, with atomic registers, deadlock freedom is violated under $JA\widetilde{\mathcal{B}}^{\bullet s}$. The following execution for two threads demonstrates this violation:

- Thread 1 moves through lines 1 through 5, setting *flag*[1] and *stage*[1] to *true*. Note that thread 1 can go through line 4 because *flag*[0] = *false*.

- Thread 0 can similarly move through lines 1 through 5; while $flag[1] = true$, we do have that $date[0] = 0 < 1 = date[1]$, so it can pass through line 4. However, on line 6, thread 0 observes $stage[1] = true$, so it has to return to line 2.

At this point, thread 0 can repeat lines 2 through 5 endlessly, as long as thread 1 does not set $stage[1]$ to *false*. Note that the resulting infinite execution satisfies $JA_{\mathcal{B}}^{\smile\bullet S}$: thread 1's read of $stage[0]$, which it has to perform on line 6, is repeatedly blocked by thread 0's writes to $stage[0]$ on lines 3 and 5.

This violation can easily be fixed by preventing a thread from endlessly repeating the loop in the entry protocol while the other thread's *stage* is *false*. This can be done by altering line 4 to instead say **await** $\forall_{j \neq i} : flag[j] = false \vee (date[i] < date[j] \wedge stage[j] = false)$. While this makes it more difficult to progress through line 4, it is impossible for all threads to get stuck there: if all threads are on line 4, then *stage* is *false* for all of them, and so the one with the lowest *date* can go to line 5. Indeed, as is shown in Table 1, with this modification Aravind's algorithm now satisfies starvation freedom under $JA_{\mathcal{B}}^{\smile\bullet S}$.

6.3 Dekker's algorithm

Dekker's algorithm originally appears in [20]. There is no clear pseudocode given there, so we use the pseudocode from [3], here given as Algorithm 2. The algorithm uses a Boolean *flag* per thread, initially *false*, and a multi-writer register *turn*, initially 0. An execution showing that Dekker's algorithm does not satisfy starvation freedom with safe registers is reported in [16]. This same execution can be found by mCRL2:

- Thread 0 goes through the algorithm without competition, and starts setting $flag[0]$ to *false* in the exit protocol, when it is currently *true*.
- Thread 1 starts the competition and reads $flag[0] = false$, the new value, on line 2. It can therefore go to the critical section, and set *turn* to 0 in the exit protocol.
- Thread 1 then starts the competition again, now reading $flag[0] = true$, the old value, on line 2. Since $turn = 0$, it goes to line 5 and starts waiting for *turn* to be 1.
- Thread 0 finishes the exit protocol and never re-attempts to enter the critical section.

Since thread 0 will never set *turn* to 1, thread 1 can never escape line 5. This execution violates deadlock freedom as well as starvation freedom, and is also applicable to regular registers. The phenomenon where two reads concurrent with the same write return first the new and then the old value is called *new-old inversion*, and is explicitly allowed by Lamport in his definitions of safe and regular registers [41]. An interesting quality of this execution is that it relies on thread 0 only finitely often executing the algorithm. If we did not define the actions nc_t for all $t \in \mathbb{T}$ to be blockable, this execution would be missed.

In [16] the following improvements are suggested to make the algorithm "RW-safe", i.e., correct with safe registers: on line 5, **await** $turn = i \vee flag[j] = false$, and on line 8, only write to *turn* if its value would be changed. Our model checking confirms that with these alterations, starvation freedom is satisfied with both safe and regular registers.

In [46], it is claimed that Dekker's algorithm without alterations is correct with non-atomic registers. Instead of dealing with the spurious violations of liveness properties via completeness criteria, they use the model checking tool UPPAAL [10] to compute the maximum number of times a thread may be overtaken by another thread. They determine that bound to be finite and conclude starvation freedom is satisfied. However, the deadlock freedom violation observed here and in [16] shows a thread never gaining access to the critical section while only being overtaken once. Hence, finding a finite upper bound to the number of overtakes is insufficient to establish deadlock freedom.

As can be observed in Table 1, both the version of Dekker's algorithm presented in [3] and

the RW-safe version from [16] are starvation-free with atomic registers under $JA^{\smile\bullet}_{\mathcal{B}}{}^{T}$, but only deadlock-free under $JA^{\smile\bullet}_{\mathcal{B}}{}^{S}$. In both variants, this is because one thread, say 0, can remain stuck on line 5 trying to perform a read, while the other thread, in this case 1, repeatedly executes the full algorithm without having to wait on thread 0, since $\mathit{flag}[0] = \mathit{false}$. In the process, thread 1 writes to the variable that thread 0 is trying to read, meaning this execution is just under $JA^{\smile\bullet}_{\mathcal{B}}{}^{S}$.

This starvation freedom violation can be easily fixed for the presentation in [3], since the variable that thread 0 is trying to read on line 5 is *turn*. If we alter the algorithm so that a thread only writes to *turn* on line 8 if this would change the value, then starvation freedom is satisfied. This change is part of the changes suggested in [16], yet that version of the algorithm is not starvation-free under $JA^{\smile\bullet}_{\mathcal{B}}{}^{S}$. This is due to the other change: on line 5, thread 0 now also has to read $\mathit{flag}[1]$, the value of which thread 1 does change every time it executes the algorithm. This violation therefore cannot be fixed so easily.

6.4 Szymanski's 3-bit linear wait algorithm

Algorithm 4 Szymanski's 3-bit linear wait algorithm

```
1:  a[i] ← true
2:  for j from 0 to N−1 do await s[j] = false
3:  w[i] ← true
4:  a[i] ← false
5:  while s[i] = false do
6:      j ← 0
7:      while j < N ∧ a[j] = false do j ← j + 1
8:      if j = N then
9:          s[i] ← true
10:         j ← 0
11:         while j < N ∧ a[j] = false do j ← j + 1
12:         if j < N then s[i] ← false
13:         else
14:             w[i] ← false
15:             for j from 0 to N − 1 do await w[j] = false
16:     if j < N then
17:         j ← 0
18:         while j < N ∧ (w[j] = true ∨ s[j] = false) do j ← j + 1
19:     if j ≠ i ∧ j < N then
20:         s[i] ← true
21:         w[i] ← false
22: for j from 0 to i − 1 do await s[j] = false
23: critical section
24: s[i] ← false
```

In [54], Szymanski proposes four mutual exclusion algorithms. Here, we discuss the first: the 3-bit linear wait algorithm, which is claimed to be correct with non-atomic registers. See Algorithm 4. Each thread has three Booleans, a, w, and s, all initially *false*. An execution showing a mutual exclusion violation for safe, regular, and atomic registers with three threads for the 3-bit linear wait algorithm is given in [51].

With two threads, we do still get a mutual exclusion violation with safe and regular

registers, given in detail in [51], but not with atomic registers. The violation specifically relies on reading $w[j]$ before $s[j]$ on line 18, so that there can be new-old inversion on $s[j]$, while still obtaining the value of $w[j]$ from before the other thread started writing to $s[j]$. We therefore considered the alternative, where $s[j]$ is read before $w[j]$ on line 18. With this change and only two threads, the algorithm satisfies all expected properties.

The observation that Szymanski's 3-bit linear wait algorithm violates mutual exclusion with three threads is also made in [45]. They suggest a different way to make the algorithm correct for two threads: instead of swapping the reads on line 18, they require a thread to also read $w[j]$ on line 22. We did not investigate this suggested change further, as we prefer the solution that does not require an additional read.

6.5 From deadlock freedom to starvation freedom

In [49, Section 2.2.2], an algorithm is presented to turn any mutual exclusion algorithm that satisfies mutual exclusion and deadlock freedom into one that satisfies starvation freedom as well. It is due to Yoah Bar-David (1998) and first appears in [55]. We take the pseudocode from [28], where it is proven that this algorithm works for safe, regular and atomic registers.

To confirm the correctness of the algorithm experimentally, we applied it to Lamport's 1-bit algorithm [39], which (by design) does not satisfy starvation freedom at all, and to the RW-safe version of Dekker's algorithm [16], where starvation freedom only fails due to writes interfering with reads. The results are given in the table with "DFtoSF". We indeed find that starvation freedom is now satisfied where previously only deadlock freedom was.

In view of space, the pseudocode is given in Appendix I.

7 Related work

To the best of our knowledge, we are the first to do automatic verification of mutual exclusion algorithms while incorporating both which operations can block each other and the effects of overlapping write operations. The two elements have been considered separately previously.

In [14], starvation-freedom of Peterson's algorithm with atomic registers is checked using mCRL2 under the justness assumption. Two different concurrency relations are considered, which are similar to our $\smile\!\!\bullet_S$ and $\smile\!\!\bullet_A$.

There have been many formal verifications of mutual exclusion algorithms with atomic registers [11, 44, 30]. Non-atomic registers have been covered less frequently, and their verification is often restricted to single-writer registers [42, 16]. The safety properties of mutual exclusion algorithms with MWMR non-atomic registers were verified using mCRL2 in [51]. In several papers by Nigro, including [45, 46], safety and liveness verification with MWMR non-atomic registers has been done using UPPAAL.

A major drawback of our approach is that we only consider a small number of parallel threads. There has been much work in the literature on parametrised verification, where the correctness of an algorithm is established for an arbitrary number of threads [13, 24, 56, 12]. Where these techniques handle liveness, it is under forms of weak or strong fairness, not justness. To our knowledge, these techniques have not yet been applied in the context of non-atomic registers. While several papers on parametrised verification, such as [2, 1], mention dropping the atomicity assumption, this refers to evaluating existential and universal conditions on all threads in a single atomic operation, rather than atomicity of the registers.

In [7, 32, 33] and other work by Hesselink, interactive theorem proving with the proof assistant PVS is used to check safety and liveness properties (under fairness) of mutual

exclusion algorithms for an arbitrary number of threads with non-atomic registers. To our knowledge, the case of writes overlapping each other has not been covered with this technique.

8 Conclusion

When it comes to analysing the correctness of algorithms, particularly when considering non-atomic registers, behavioural reasoning is often insufficient. Mistakes can be subtle, and may depend on edge-cases that are easily overlooked. Model checking is a solution here; we formally model the threads executing the algorithm, as well as the registers through which they communicate, and the entire state-space is searched for possibly violations of correctness properties. In this work, we verify a large number of mutual exclusion algorithms using the model checking toolset mCRL2. We expand on previous work by checking liveness properties – deadlock freedom and starvation freedom – in addition to the main safety property. To circumvent spurious violating paths in our models, we incorporate the completeness criterion justness into our verifications. We checked algorithms under six different memory models, where a memory model is a combination of a register model (safe, regular, or atomic) and a model of which register access operations can block each other. The former dimension we capture in the models themselves, by modelling the behaviour of the three types of register. The latter, we capture in the concurrency relations employed as part of justness. We found a number of interesting violations of correctness properties, and in some cases could suggest improvements to algorithms to fix these violations. We find that there are several algorithms that satisfy all three properties for four out of six memory models. For three threads, this is accomplished by the fixed version of Aravind's BLRU algorithm and Lamport's 3-bit algorithm. If we also consider algorithms for just two threads, then Anderson's algorithm and the fixed version of Szymanski's 3-bit linear wait algorithm also meet this bar. We also considered an algorithm to turn deadlock-free algorithms into starvation-free ones, and experimentally confirmed that it indeed works for Dekker's algorithm made RW-safe and Lamport's 1-bit algorithm.

References

1 Parosh Aziz Abdulla, Frédéric Haziza, and Lukáš Holík. Parameterized verification through view abstraction. *International Journal on Software Tools for Technology Transfer*, 18(5):495–516, 2016. doi:10.1007/s10009-015-0406-x.

2 Parosh Aziz Abdulla, Noomene Ben Henda, Giorgio Delzanno, and Ahmed Rezine. Handling parameterized systems with non-atomic global conditions. In Francesco Logozzo, Doron A. Peled, and Lenore D. Zuck, editors, *Verification, Model Checking, and Abstract Interpretation* (VMCAI'08), volume 4905 of *Lecture Notes in Computer Science*, pages 22–36. Springer, 2008. doi:10.1007/978-3-540-78163-9_7.

3 K. Alagarsamy. Some myths about famous mutual exclusion algorithms. *SIGACT News*, 34(3):94–103, 2003. doi:10.1145/945526.945527.

4 James H. Anderson. A fine-grained solution to the mutual exclusion problem. *Acta Informatica*, 30(3):249–265, 1993. doi:10.1007/BF01179373.

5 Krzysztof R. Apt and Ernst-Rüdiger Olderog. Proof rules and transformations dealing with fairness. *Science of Computer Programming*, 3(1):65–100, 1983. doi:10.1016/0167-6423(83)90004-7.

6 Alex A. Aravind. Yet another simple solution for the concurrent programming control problem. *IEEE Transactions on Parallel and Distributed Systems*, 22(6):1056–1063, 2011. doi:10.1109/TPDS.2010.172.

7 Alex A. Aravind and Wim H. Hesselink. Nonatomic dual bakery algorithm with bounded tokens. *Acta Informatica*, 48(2):67–96, 2011. doi:10.1007/s00236-011-0132-0.

8 Hagit Attiya, Rachid Guerraoui, Danny Hendler, Petr Kuznetsov, Maged M. Michael, and Martin T. Vechev. Laws of order: expensive synchronization in concurrent algorithms cannot be eliminated. In Thomas Ball and Mooly Sagiv, editors, Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on *Principles of Programming Languages* (POPL'11), pages 487–498. ACM, 2011. doi:10.1145/1926385.1926442.

9 Hagit Attiya and Jennifer L. Welch. *Distributed computing - fundamentals, simulations, and advanced topics* (2nd ed.). Wiley series on parallel and distributed computing. Wiley, 2004.

10 Gerd Behrmann, Alexandre David, and Kim Guldstrand Larsen. A tutorial on uppaal. In Marco Bernardo and Flavio Corradini, editors, *Formal Methods for the Design of Real-Time Systems,* International School on Formal Methods for the Design of Computer, Communication and Software Systems, (SFM-RT'04), volume 3185 of *Lecture Notes in Computer Science*, pages 200–236. Springer, 2004. doi:10.1007/978-3-540-30080-9_7.

11 Mordechai Ben-Ari. *Principles of the Spin model checker*. Springer, 2008. doi:10.1007/978-1-84628-770-1.

12 Roderick Bloem, Swen Jacobs, Ayrat Khalimov, Igor Konnov, Sasha Rubin, Helmut Veith, and Josef Widder. Decidability in parameterized verification. *SIGACT News*, 47(2):53–64, 2016. doi:10.1145/2951860.2951873.

13 Ahmed Bouajjani, Bengt Jonsson, Marcus Nilsson, and Tayssir Touili. Regular model checking. In E. Allen Emerson and A. Prasad Sistla, editors, *Computer Aided Verification* (CAV'00), volume 1855 of *Lecture Notes in Computer Science*, pages 403–418. Springer, 2000. doi:10.1007/10722167_31.

14 Mark Bouwman, Bas Luttik, and Tim A. C. Willemse. Off-the-shelf automated analysis of liveness properties for just paths. *Acta Informatica*, 57(3-5):551–590, 2020. doi:10.1007/s00236-020-00371-w.

15 Peter A. Buhr, David Dice, and Wim H. Hesselink. High-performance N-thread software solutions for mutual exclusion. *Concurrency and Computation: Practice and Experience*, 27(3):651–701, 2015. doi:10.1002/cpe.3263.

16 Peter A. Buhr, David Dice, and Wim H. Hesselink. Dekker's mutual exclusion algorithm made RW-safe. *Concurrency and Computation: Practice and Experience*, 28(1):144–165, 2016. doi:10.1002/cpe.3659.

17 Olav Bunte, Jan Friso Groote, Jeroen J. A. Keiren, Maurice Laveaux, Thomas Neele, Erik P. de Vink, Wieger Wesselink, Anton Wijs, and Tim A. C. Willemse. The mCRL2 toolset for analysing concurrent systems – improvements in expressivity and usability. In Tomás Vojnar and Lijun Zhang, editors, *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), Part II, volume 11428 of *Lecture Notes in Computer Science*, pages 21–39. Springer, 2019. doi:10.1007/978-3-030-17465-1_2.

18 James E. Burns and Nancy A. Lynch. Bounds on shared memory for mutual exclusion. *Information and Computation*, 107(2):171–184, 1993. doi:10.1006/inco.1993.1065.

19 Flavio Corradini, Maria Rita Di Berardini, and Walter Vogler. Time and fairness in a process algebra with non-blocking reading. In Mogens Nielsen, Antonín Kucera, Peter Bro Miltersen, Catuscia Palamidessi, Petr Tuma, and Frank D. Valencia, editors, *SOFSEM'09: Theory and Practice of Computer Science, 35th Conference on Current Trends in Theory and Practice of Computer Science*, volume 5404 of *Lecture Notes in Computer Science*, pages 193–204. Springer, 2009. doi:10.1007/978-3-540-95891-8_20.

20 Edsger W Dijkstra. Over de sequentialiteit van procesbeschrijvingen (ewd-35). *Center for American History, University of Texas at Austin*, 1962. URL: http://www.cs.utexas.edu/users/EWD/ewd00xx/EWD35.PDF.

21 Edsger W. Dijkstra. Solution of a problem in concurrent programming control. *Communications of the ACM*, 8(9):569, 1965. doi:10.1145/365559.365617.

22 Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. Patterns in property specifications for finite-state verification. In Barry W. Boehm, David Garlan, and Jeff Kramer, editors, Proceedings of the 1999 International Conference on *Software Engineering* (ICSE'99), pages 411–420. ACM, 1999. doi:10.1145/302405.302672.

23 Victor Dyseryn, Rob J. van Glabbeek, and Peter Höfner. Analysing mutual exclusion using process algebra with signals. In Kirstin Peters and Simone Tini, editors, Proceedings Combined 24th International Workshop on *Expressiveness in Concurrency* and 14th Workshop on *Structural Operational Semantics* (EXPRESS/SOS'17), volume 255 of *Electronic Proceedings in Theoretical Computer Science*, pages 18–34, 2017. doi:10.4204/EPTCS.255.2.

24 Dana Fisman and Amir Pnueli. Beyond regular model checking. In Ramesh Hariharan, Madhavan Mukund, and V. Vinay, editors, *Foundations of Software Technology and Theoretical Computer Science* (FST&TCS'01), volume 2245 of *Lecture Notes in Computer Science*, pages 156–170. Springer, 2001. doi:10.1007/3-540-45294-X_14.

25 Rob J. van Glabbeek. Is speed-independent mutual exclusion implementable? (Invited talk). In Sven Schewe and Lijun Zhang, editors, 29th International Conference on *Concurrency Theory (CONCUR'18)*, volume 118 of *LIPIcs*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018. doi:10.4230/LIPIcs.CONCUR.2018.3.

26 Rob J. van Glabbeek. Justness - A completeness criterion for capturing liveness properties (extended abstract). In Mikolaj Bojanczyk and Alex Simpson, editors, *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), volume 11425 of *Lecture Notes in Computer Science*, pages 505–522. Springer, 2019. doi:10.1007/978-3-030-17127-8_29.

27 Rob J. van Glabbeek. Modelling mutual exclusion in a process algebra with time-outs. *Information and Computation*, 294:105079, 2023. doi:10.1016/j.ic.2023.105079.

28 Rob J. van Glabbeek and Daniele Gorla. On the notions of bounded bypass, and how to make any deadlock-free mutex protocol satisfy one of them. 2025. To appear.

29 Rob J. van Glabbeek and Peter Höfner. Progress, justness, and fairness. *ACM Computing Surveys*, 52(4), 2019. doi:10.1145/3329125.

30 Jan Friso Groote and Jeroen J. A. Keiren. Tutorial: Designing distributed software in mCRL2. In Kirstin Peters and Tim A. C. Willemse, editors, Proceedings 41st IFIP WG 6.1 International Conference on *Formal Techniques for Distributed Objects, Components, and Systems* (FORTE'21), held as part of the 16th International Federated Conference on *Distributed Computing Techniques* (DisCoTec 2021), volume 12719, pages 226–243. Springer, 2021. doi:10.1007/978-3-030-78089-0_15.

31 Jan Friso Groote and Mohammad Reza Mousavi. *Modeling and Analysis of Communicating Systems*. MIT Press, 2014. doi:10.7551/mitpress/9946.001.0001.

32 Wim H. Hesselink. Mechanical verification of lamport's bakery algorithm. *Science of Computer Programming*, 78(9):1622–1638, 2013. doi:10.1016/j.scico.2013.03.003.

33 Wim H. Hesselink. Mutual exclusion by four shared bits with not more than quadratic complexity. *Science of Computer Programming*, 102:57–75, 2015. doi:10.1016/j.scico.2015.01.001.

34 C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, Englewood Cliffs, 1985.

35 Joep L. W. Kessels. Arbitration without common modifiable variables. *Acta Informatica*, 17:135–141, 1982. doi:10.1007/BF00288966.

36 Donald E. Knuth. Additional comments on a problem in concurrent programming control. *Communications of the ACM*, 9(5):321–322, 1966. doi:10.1145/355592.365595.

37 Leslie Lamport. A new solution of dijkstra's concurrent programming problem. *Communications of the ACM*, 17(8):453–455, 1974. doi:10.1145/361082.361093.

38 Leslie Lamport. The mutual exclusion problem: Part I—a theory of interprocess communication. *J. ACM*, 33(2):313–326, 1986. doi:10.1145/5383.5384.

39 Leslie Lamport. The mutual exclusion problem: Part II—statement and solutions. *J. ACM*, 33(2):327–348, 1986. doi:10.1145/5383.5385.

40 Leslie Lamport. On interprocess communication. Part I: basic formalism. *Distributed Comput.*, 1(2):77–85, 1986. doi:10.1007/BF01786227.

41 Leslie Lamport. On interprocess communication. Part II: algorithms. *Distributed Comput.*, 1(2):86–101, 1986. doi:10.1007/BF01786228.

42 Leslie Lamport. *The TLA+ Hyperbook*, chapter 7.8.4: The Real Bakery Algorithm. 2015. Available at http://lamport.azurewebsites.net/tla/hyperbook.html, accessed on 26 April 2023.

43 Daniel Lehmann, Amir Pnueli, and Jonathan Stavi. Impartiality, justice and fairness: The ethics of concurrent termination. In Shimon Even and Oded Kariv, editors, *Automata, Languages and Programming* (ICALP'81), volume 115 of *Lecture Notes in Computer Science*, pages 264–277. Springer, 1981. doi:10.1007/3-540-10843-2_22.

44 Radu Mateescu and Wendelin Serwe. Model checking and performance evaluation with CADP illustrated on shared-memory mutual exclusion protocols. *Science of Computer Programming*, 78(7):843–861, 2013. Special section on Formal Methods for Industrial Critical Systems (FMICS 2009 + FMICS 2010) & Special section on Object-Oriented Programming and Systems (OOPS 2009), a special track at the 24th ACM Symposium on Applied Computing. doi:10.1016/j.scico.2012.01.003.

45 Libero Nigro. Verifying mutual exclusion algorithms with non-atomic registers. *Algorithms*, 17(12), 2024. doi:10.3390/a17120536.

46 Libero Nigro, Franco Cicirelli, and Francesco Pupo. Modeling and analysis of Dekker-based mutual exclusion algorithms. *Computers*, 13(6):133, 2024. doi:10.3390/computers13060133.

47 Gary L. Peterson. Myths about the mutual exclusion problem. *Information Processing Letters*, 12(3):115–116, 1981. doi:10.1016/0020-0190(81)90106-X.

48 Gary L. Peterson. A new solution to lamport's concurrent programming problem using small shared variables. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 5(1):56–65, 1983. doi:10.1145/357195.357199.

49 Michel Raynal. *Concurrent Programming - Algorithms, Principles, and Foundations*. Springer, 2013. doi:10.1007/978-3-642-32027-9.

50 Cheng Shao, Jennifer L. Welch, Evelyn Pierce, and Hyunyoung Lee. Multiwriter consistency conditions for shared memory registers. *SIAM Journal on Computing*, 40(1):28–62, 2011. doi:10.1137/07071158X.

51 Myrthe S. C. Spronck and Bas Luttik. Process-algebraic models of multi-writer multi-reader non-atomic registers. In Guillermo A. Pérez and Jean-François Raskin, editors, 34th International Conference on *Concurrency Theory* (CONCUR'23), volume 279 of *LIPIcs*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023. Full version available at https://arxiv.org/abs/2307.05143. doi:10.4230/LIPIcs.CONCUR.2023.5.

52 Myrthe S. C. Spronck, Bas Luttik, and Tim A. C. Willemse. Progress, justness and fairness in modal μ-calculus formulae. In Rupak Majumdar and Alexandra Silva, editors, 35th International Conference on *Concurrency Theory* (CONCUR'24), volume 311 of *LIPIcs*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2024. doi:10.4230/LIPIcs.CONCUR.2024.38.

53 Boleslaw K. Szymanski. A simple solution to lamport's concurrent programming problem with linear wait. In Jacques Lenfant, editor, Proceedings of the 2nd international conference on *Supercomputing* (ICS'88), pages 621–626. ACM, 1988. doi:10.1145/55364.55425.

54 Boleslaw K. Szymanski. Mutual exclusion revisited. In Joshua Maor and Abraham Peled, editors, *Next Decade in Information Technology:* Proceedings of the 5th *Jerusalem Conference on Information Technology*, pages 110–117. IEEE Computer Society, 1990. doi:10.1109/JCIT.1990.128275.

55 Gadi Taubenfeld. *Synchronization algorithms and concurrent programming*. Pearson Education, 2006.

56 Lenore D. Zuck and Amir Pnueli. Model checking and abstraction to the aid of parameterized systems (a survey). *Computer Languages, Systems & Structures*, 30(3):139–169, 2004. doi:10.1016/j.cl.2004.02.006.

A Safe, regular and atomic SWMR registers (an example)

We illustrate the differences between Lamport's safe, regular and atomic SWMR registers with an example. Consider the execution illustrated in Figure 1, which has three threads, with id's 0, 1 and 2, operating on a register x with domain $\{0, 1, 2\}$.

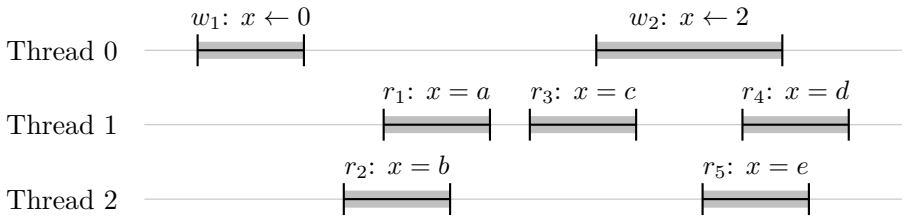


Figure 1 Example behaviour of SWMR safe, regular and atomic registers.

Let us consider which values are possible for a, b, c, d and e, depending on which type of register x is.

Both r_1 and r_2 are not concurrent with any write, hence both reads will return the last written value of the register, regardless of the type of x. Therefore, we have $a = 0$ and $b = 0$ for safe, regular and atomic registers.

If x is a safe register, then c, d and e may all be any value of $0, 1$ or 2. These three assignments are independent of each other, so all 27 different possible combinations of choices for c, d and e are allowed.

If x is a regular register, then c, d and e may all be any value of 0 or 2: the old value or the value that is being written concurrently. Here too, the three returned values are independent, so there are 8 different possibilities. In particular, it can be that c=2 but d=0; this possibility is called new-old inversion.

If x is an atomic register, then the values returned by the reads must reflect some order on all operations in the execution, and this order must respect the real-time ordering of the operations. We interpret the figure as showing real time, meaning that, for example, r_1 must be ordered before w_2 since r_1 ends before w_2 begins. Consequently, there are only five combinations of values of c, d and e allowed. If c is 0, then any combination of d and e being 0 or 2 is accepted, since r_3 can be ordered before w_2 without affecting whether r_4 and r_5 are before or after w_2. If instead we have $c = 2$, then r_3 must be ordered after w_2 and hence r_4 and r_5, which in real time come after r_3, must also be ordered after r_2, giving us $d = e = 2$.

B Register models

In this appendix, we expand on the material of Section 3, by giving the formal process-algebraic models of the registers. First, we present the general structure of such a model and explain how to translate it to an LTS. We then give the three models, as well as the definitions of all the access and update functions. We leave the status object abstract: it is not necessary to define the data structure itself, as long as it is clear what information can be retrieved from it.

B.1 Structure and functions

Recall that we use $\mathbb{T}$ for the thread identifiers, $\mathbb{R}$ for register identifiers, and that for every $r \in \mathbb{R}$, the set $\mathbb{D}_r$ contains all values that r may hold. Additionally, we use a status object as the finite memory of a register, the set of possible statuses being $\mathbb{S}$.

We define the following structure, shared by all three register models. Let $\gamma \in \{saf, reg, ato\}$, then each model looks as follows, for some number n:

$$Reg_\gamma(r : \mathbb{R}, s : \mathbb{S}) = \sum_{t \in \mathbb{T}} \sum_{d \in \mathbb{D}_r} \sum_{0 \leq j < n} (c_j(s,t,d) \to a_j(t,d) \cdot Reg_\gamma(r, u_j(s,t,d))) \quad (1)$$

This represents a register with id r, that tracks its status with s. The process first sums over $t \in \mathbb{T}$ and $d \in \mathbb{D}_r$, allowing interaction by all threads and with all possible data parameters. Furthermore, it has n summands, each of the form $c_j(s,t,d) \to a_j(t,d) \cdot Reg_\gamma(r, u_j(s,t,d))$ where $c_j(s,t,d)$ is a Boolean condition, $a_j(t,d)$ is an action, and u_j is an *update function* that takes s, t and d and returns the updated status object $s' \in \mathbb{S}$. Such a process equation gives rise to an LTS. Given a predefined initial state $init \in \mathbb{S}$, the LTS of $Reg_\gamma(r, init)$ is:

$$(\mathbb{S}, \bigcup_{0 \leq j < n} \{a_j(t,d) \mid t \in \mathbb{T}, d \in \mathbb{D}_r\}, init, \bigcup_{0 \leq j < n} \{(s, a_j(t,d), u_j(s,t,d)) \mid t \in \mathbb{T}, d \in \mathbb{D}_r \wedge c_j(s,t,d)\}) \quad (2)$$

We now describe the initial state and the update functions. As stated above, we do not wish the go into the implementation details of the status object. Instead, we define a collection of *access functions* which retrieve information from the current state. The initial state, as well as the effects of the update functions, are then defined by how they alter the results of the access functions. We use the following access functions, which are local to any given register r:

- $stor : \mathbb{S} \to \mathbb{D}_r$, the value that is currently stored in the register.
- $rds : \mathbb{S} \to 2^{\mathbb{T}}$, the set of thread id's of threads that have invoked a read operation on this register that has not yet had its response.
- $wrts : \mathbb{S} \to 2^{\mathbb{T}}$, the set of thread id's of threads that have invoked a write operation on this register that has not yet had its response.
- $pend : \mathbb{S} \to 2^{\mathbb{T}}$, the set of thread id's of threads that have invoked an operation that has not been ordered yet. Only used by the regular and atomic models.
- $rec : \mathbb{S} \times \mathbb{T} \to \mathbb{D}_r$, a mapping that allows us to record a single data value per thread, for instance used to remember what value was passed with a start write action.
- The predicate $ovrl$ on $\mathbb{S} \times \mathbb{T}$, which stores whether an ongoing read or write operation of a thread has encountered an overlapping write. Only used by the safe model.
- $posv : \mathbb{S} \times \mathbb{T} \to 2^{\mathbb{D}_r}$, a mapping that stores a set of values per thread, representing the possible return values of an ongoing read by a thread. Only used by the regular model.

The initial state $init$ is defined as follows, for all $t \in \mathbb{T}$ and a pre-defined initial value d_{init}:

$$\begin{array}{llll} stor(init) = d_{init} & wrts(init) = \emptyset & rec(init,t) = d_{init} & posv(init,t) = \emptyset \\ rds(init) = \emptyset & pend(init) = \emptyset & ovrl(init,t) = false & \end{array}$$

The initial value d_{init} of a register depends on the modelled algorithm.

We now define the update functions in a similar way to the initial state: by showing how the return values of the access functions are altered by the update function. Each update function corresponds to an action and is applied when that action occurs; if the action's name is a, the update function is called ua. Not every update function uses the data parameter that is passed to it according to (1); in these cases we only give the thread id and status parameters. If an access function is not mentioned, then its return value after the update is the same as before. Given an arbitrary state $s \in \mathbb{S}$, thread id $t \in \mathbb{T}$ and data value $d \in \mathbb{D}_r$:

If $s' = usr(s,t)$, then:

$$\begin{aligned} rds(s') &= rds(s) \cup \{t\} \\ pend(s') &= pend(s) \cup \{t\} \\ ovrl(s',t) &= (wrts(s) > 0) \\ posv(s',t) &= \{stor(s)\} \cup \{d' \mid \exists_{t' \in wrts(s)}.rec(s,t') = d'\} \end{aligned}$$

If $s' = ufr(s,t)$, then:

$$rds(s') = rds(s) \setminus \{t\}$$

If $s' = usw(s,t,d)$, then for all $t' \neq t$:

$$\begin{aligned} wrts(s') &= wrts(s) \cup \{t\} \\ pend(s') &= pend(s) \cup \{t\} \\ rec(s',t) &= d \\ ovrl(s',t) &= (wrts(s) > 0) \\ ovrl(s',t') &= true \\ posv(s',t') &= posv(s,t') \cup \{d\} \end{aligned}$$

If $s' = ufw(s,t,d)$, then:

$$\begin{aligned} stor(s') &= d \\ wrts(s') &= wrts(s) \setminus \{t\} \end{aligned}$$

If $s' = uor(s,t)$, then:

$$\begin{aligned} pend(s') &= pend(s) \setminus \{t\} \\ rec(s',t) &= stor(s) \end{aligned}$$

If $s' = uow(s,t,d)$, then:

$$\begin{aligned} stor(s') &= d \\ pend(s') &= pend(s) \setminus \{t\} \end{aligned}$$

These formal definitions correspond to the intuitive descriptions given above. Of note is that $ovrl(s',t')$ is set to *true* whenever a thread $t \neq t'$ starts a write, even if t' is not actively reading or writing. This is done for simplicity of the definition: when t' starts reading or writing, it will reset its own *ovrl* to the correct value, depending on whether there is an overlapping write active at that point. Something similar is done with *posv*: when a write is started, its value gets added to the *posv* sets of every other thread, even if they are not actively reading. When a thread starts reading, it sets its own *posv* correctly.

We now give the three register models.

B.2 Safe MWMR registers

See Figure 2 for the process equation representing our safe register model.

The correspondence between the process and the four rules given for MWMR safe registers in Section 3.1 is rather direct: the first two summands allow a thread that is not currently reading or writing to begin a read or a write, and the remaining four each represent one of the four rules, in order. Note that, in the case of a finish write without overlap, we use *rec* to retrieve which value this thread intended to write so that the register state can be appropriately updated.

$$Reg_{saf}(r:\mathbb{R}, s:\mathbb{S}) = \sum_{t\in\mathbb{T}}\sum_{d\in\mathbb{D}_r}\left(\begin{array}{ll} & (t \notin (rds(s) \cup wrts(s))) \to sr_{t,r} \cdot Reg_{saf}(r, usr(s,t)) \\ + & (t \notin (rds(s) \cup wrts(s))) \to sw_{t,r}(d) \cdot Reg_{saf}(r, usw(s,t,d)) \\ + & (t \in rds(s) \wedge \neg ovrl(s,t)) \to fr_{t,r}(stor(s)) \cdot Reg_{saf}(r, ufr(s,t)) \\ + & (t \in rds(s) \wedge ovrl(s,t)) \to fr_{t,r}(d) \cdot Reg_{saf}(r, ufr(s,t)) \\ + & (t \in wrts(s) \wedge \neg ovrl(s,t)) \to fw_{t,r} \cdot Reg_{saf}(r, ufw(s,t,rec(s,t))) \\ + & (t \in wrts(s) \wedge ovrl(s,t)) \to fw_{t,r} \cdot Reg_{saf}(r, ufw(s,t,d)) \end{array}\right)$$

Figure 2 Safe register process

B.3 Regular MWMR registers

See Figure 3 for the process equation representing our regular register model. Recall that regular registers use the order write action to generate a global ordering on all write operations on a register on the fly.

$$Reg_{reg}(r:\mathbb{R}, s:\mathbb{S}) = \sum_{t\in\mathbb{T}}\sum_{d\in\mathbb{D}_r}\left(\begin{array}{ll} & (t \notin (rds(s) \cup wrts(s))) \to sr_{t,r} \cdot Reg_{reg}(r, usr(s,t)) \\ + & (t \notin (rds(s) \cup wrts(s))) \to sw_{t,r}(d) \cdot Reg_{reg}(r, usw(s,t,d)) \\ + & (t \in rds(s) \wedge d \in posv(s,t)) \to fr_{t,r}(d) \cdot Reg_{reg}(r, ufr(s,t)) \\ + & (t \in wrts(s) \wedge t \in pend(s)) \to ow_{t,r} \cdot Reg_{reg}(r, uow(s,t,rec(s,t))) \\ + & (t \in wrts(s) \wedge t \notin pend(s)) \to fw_{t,r} \cdot Reg_{reg}(r, ufw(s,t,stor(s))) \end{array}\right)$$

Figure 3 Regular register process

Similar to the safe register process, the first two summands are merely allowing an idle thread to begin a read or write operation. The third summand corresponds to finishing a read by returning a value that is in the set of possible values to be returned for this read. Recall that, by the definition of *posv*, this set is constructed as follows: when a read starts, the set is initialised to the current stored value of the register and the intended write value of every active write. Subsequently, whenever a write occurs, its intended value is added to the set. This way, at the finish read, the set will contain exactly those values that the read could return. The fourth summand allows the occurrence of the ordering action; at this time the intended value of the write, which was temporary stored in the access function *rec*, is logged as the stored value of the register. The final summand describes the ending of a write operation. The *ufw* update function will set the stored value to whatever data value is passed. In this case, the stored value should not change at the finish write, since it was already changed at the order write. Hence, we simply pass $stor(s)$. Note that we use *pend* to determine if the order write action still has to occur.

B.4 Atomic MWMR registers

See Figure 4 for our model of MWMR atomic registers. Recall that, in addition to the order write action, the atomic register model also uses the order read action. This way, it generates an ordering on all operations on a register.

$$Reg_{ato}(r:\mathbb{R}, s:\mathbb{S}) = \sum_{t\in\mathbb{T}}\sum_{d\in\mathbb{D}_r}\left(\begin{array}{ll} & (t\notin(rds(s)\cup wrts(s))) \to sr_{t,r}\cdot Reg_{ato}(r, usr(s,t)) \\ + & (t\notin(rds(s)\cup wrts(s))) \to sw_{t,r}(d)\cdot Reg_{ato}(r, usw(s,t,d)) \\ + & (t\in rds(s)\wedge t\in pend(s)) \to or_{t,r}\cdot Reg_{ato}(r, uor(s,t)) \\ + & (t\in wrts(s)\wedge t\in pend(s)) \to ow_{t,r}\cdot Reg_{ato}(r, uow(s,t,rec(s,t))) \\ + & (t\in rds(s)\wedge t\notin pend(s)) \to fr_{t,r}(rec(s,t))\cdot Reg_{ato}(r, ufr(s,t)) \\ + & (t\in wrts(s)\wedge t\notin pend(s)) \to fw_{t,r}\cdot Reg_{ato}(r, ufw(s,t,stor(s))) \end{array}\right)$$

Figure 4 Atomic register process

Summands one, three and five are the invocation, ordering and response of a read operation respectively. Similarly, summands two, four and six are the invocation, ordering and response of a write operation. We use *pend* to determine whether an operation's order action still has to occur. When a read is ordered, we save the current stored value of the register in *rec*, so that this value can be returned when the read ends. For writes, we already update *stor* when the write is ordered, meaning that when the write ends we do not want to change *stor* further and just pass the current value back.

C Thread consistency proof

In this appendix, we prove Lemma 6. We first provide an alternate perspective on the definitions of the access functions given in Appendix B, which make it clearer how a transition affects the status object. We then prove a supporting lemma, and finally prove Lemma 6 itself.

When it comes to applying the definitions of the access functions, we find it easier to reason from the perspective of transitions: given a transition $s \xrightarrow{a} s'$ in an LTS that is derived from a register model as described in Appendix B, the relation of the output of an access function on s' to the output on s, depending on the action a. This perspective can be derived directly from the definitions of the update functions, and the tight coupling between update functions and actions.

▶ **Observation 13.** Consider the LTS $(\mathcal{S}, Act, init, Trans)$ associated with a register $r\in\mathbb{R}$. Let $s, s'\in\mathcal{S}$ and $a\in Act$ such that $s\xrightarrow{a}s'$, and let $t\in\mathbb{T}$. Then the following equations hold:

$$stor(s') = \begin{cases} d & \text{if } a = fw_{t,r} \text{ and } s' = ufw(s,t,d) \text{ for } t\in\mathbb{T}, d\in\mathbb{D}_r \\ d & \text{if } a = ow_{t,r} \text{ and } s' = uow(s,t,d) \text{ for } t\in\mathbb{T}, d\in\mathbb{D}_r \\ stor(s) & \text{otherwise} \end{cases}$$

$$rds(s') = \begin{cases} rds(s)\cup\{t\} & \text{if } a = sr_{t,r} \text{ for } t\in\mathbb{T} \\ rds(s)\setminus\{t\} & \text{if } a = fr_{t,r}(d) \text{ for } t\in\mathbb{T}, d\in\mathbb{D}_r \\ rds(s) & \text{otherwise} \end{cases}$$

$$wrts(s') = \begin{cases} wrts(s)\cup\{t\} & \text{if } a = sw_{t,r}(d) \text{ for } t\in\mathbb{T}, d\in\mathbb{D}_r \\ wrts(s)\setminus\{t\} & \text{if } a = fw_{t,r} \text{ for } t\in\mathbb{T} \\ wrts(s) & \text{otherwise} \end{cases}$$

$$pend(s') = \begin{cases} pend(s) \cup \{t\} & \text{if } a = sr_{t,r} \text{ or } a = sw_{t,r}(d) \text{ for } t \in \mathbb{T}, d \in \mathbb{D}_r \\ pend(s) \setminus \{t\} & \text{if } a = or_{t,r} \text{ or } a = ow_{t,r} \text{ for } t \in \mathbb{T} \\ pend(s) & \text{otherwise} \end{cases}$$

$$rec(s', t) = \begin{cases} d & \text{if } a = sw_{t,r}(d) \text{ for } d \in \mathbb{D}_r \\ stor(s) & \text{if } a = or_{t,r} \\ rec(s, t) & \text{otherwise} \end{cases}$$

$$ovrl(s', t) = \begin{cases} (wrts(s) > 0) & \text{if } a = sr_{t,r} \text{ or } a = sw_{t,r}(d) \text{ for } d \in \mathbb{D}_r \\ true & \text{if } a = sw_{t',r}(d) \text{ for } t' \in \mathbb{T}, t' \neq t \text{ and } d \in \mathbb{D}_r \\ ovrl(s, t) & \text{otherwise} \end{cases}$$

$$posv(s', t) = \begin{cases} \{stor(s)\} \cup \{d \mid \exists_{t' \in wrts(s)} : rec(s, t') = d\} & \text{if } a = sr_{t,r} \\ posv(s, t) \cup \{d\} & \text{if } a = sw_{t',r}(d) \text{ for } t' \in \mathbb{T}, t \neq t', d \in \mathbb{D}_r \\ posv(s, t) & \text{otherwise} \end{cases}$$

We use the following lemma in our proof of Lemma 6.

▶ **Lemma 14.** Let s be the state of a safe register that is reachable from its initial state, and let $t \in \mathbb{T}$ be a thread identifier. If $t \in rds(s)$ and $wrts(s) \neq \emptyset$, then $ovrl(s, t)$.

Proof. Let r be a safe register process with LTS $(\mathcal{S}, Act, init, Trans)$ and let $s \in \mathcal{S}$ be a state reachable from $init$. Let t be an element of $\mathbb{T}$ and assume that $t \in rds(s)$ and $wrts(s) \neq \emptyset$. Then there exists $t' \in \mathbb{T}$ such that $t' \in wrts(s)$. Since a thread cannot read and write simultaneously, we know that $t \neq t'$. We prove $ovrl(s, t)$. Recall that $rds(init) = wrts(init) = \emptyset$. Since $t \in rds(s)$ and $t' \in wrts(s)$, it follows from the definitions of rds and $wrts$ that $s \neq init$ and that every path from $init$ to s must contain occurrences of $sr_{t,r}$ and $sw_{t',r}(d)$ for some $d \in \mathbb{D}_r$. Let π be an arbitrary path from $init$ to s, and consider the last occurrence of $sr_{t,r}$ on π, and the last occurrence of $sw_{t',r}(d)$ for any $d \in \mathbb{D}_r$ on π. If after $sr_{t,r}$ there were an occurrence of $fr_{t,r}(d')$ for some $d' \in \mathbb{D}_r$ along π, then $t \notin rds(s)$, so there is no such occurrence. Similarly, if there were an occurrence of $fw_{t',r}$ after $sw_{t',r}(d)$ on π, then $t' \notin wrts(s)$, so there is no such $fw_{t',r}$. We do a case distinction on whether $sr_{t,r}$ occurs before $sw_{t',r}(d)$ or vice versa.

- Suppose $sr_{t,r}$ is before $sw_{t',r}(d)$. Let $s_b sw_{t',r}(d) s_a$ be the fragment π with the last occurrence of $sw_{t',r}(d)$. Then $ovrl(s_a, t) = true$ according to the definition of $ovrl$ as given in Observation 13. Assume towards a contradiction that $\neg ovrl(s, t)$ is true. Then on the suffix π' of π between s_a and s, there must be an occurrence of $sr_{t,r}$ or $sw_{t,r}(d')$ for some $d' \in \mathbb{D}_r$ from a state s_c such that $wrts(s_c) = \emptyset$. However, since there is no $fw_{t',r}$ on π', $t' \in wrts(s_c)$ for all states s_c on π'. Hence, we have reached a contradiction and $ovrl(s, t)$ must be true.
- Suppose $sw_{t',r}(d)$ is before $sr_{t,r}$. Let $s_b sr_{t,r} s_a$ be the fragment of π with the last occurrence of $sr_{t,r}$. Since there is no $fw_{t',r}$ after $sw_{t',r}(d)$ on π, $t' \in wrts(s_b)$. It follows from Observation 13 that $ovrl(s_a, t)$. Assume towards a contradiction that $\neg ovrl(s, t)$ is true. Then on the suffix π' of π between s_a and s, $sr_{t,r}$ or $sw_{t,r}(d')$ occurs for some $d' \in \mathbb{D}_r$ from a state s_c such that $wrts(s_c) = \emptyset$. But since $t' \in wrts(s_c)$ for all s_c along π', we have a contradiction. Therefore, $ovrl(s, t)$ must be true.

In both cases, we have shown that $ovrl(s, t)$. ◀

We now provide the proof of Lemma 6.

▶ **Lemma 6.** Let $M = (\mathcal{S}, Act, init, Trans, thr, reg)$ be a thread-register model. Then the LTS $(\mathcal{S}, Act, init, Trans)$ is thread consistent with respect to the mapping thr.

Proof. Let $M = (\mathcal{S}, Act, init, Trans, thr, reg)$ be a thread-register model constructed as the parallel composition $P_1 \parallel \ldots \parallel P_k$ with $k = |\mathbb{T}| + |\mathbb{R}|$, where P_1 to $P_{|\mathbb{T}|}$ are thread LTSs and $P_{|\mathbb{T}|+1}$ to P_k are register LTSs. Let $P_x = (\mathcal{S}_x, Act_x, init_x, Trans_x)$ for all $1 \leq x \leq k$. Let $\#$ be a mapping from thread and register id's to natural numbers, which yields the index of that thread or register in the parallel composition.

Let $s = (s_1, \ldots, s_k)$ be a state in $\mathcal{S}$ and a an action in Act such that a is enabled in s and there exists an action $b \in Act$ and a state $s' = (s'_1, \ldots, s'_k) \in \mathcal{S}$ such that $thr(a) \neq thr(b)$ and $(s, b, s') \in Trans$. We prove that a is enabled in s'.

Let $thr(a) = t$ and $reg(a) = r$, with $t \in \mathbb{T}$ and $r \in \mathbb{R} \cup \{\bot\}$. From the definition of parallel composition, specifically the construction of $Trans$, it follows that the target state of (s, b, s') can only differ from s for the parallel components that are involved in b: $P_{\#(thr(b))}$ and, if $reg(b) \neq \bot$, $P_{\#(reg(b))}$. Since $thr(b) \neq t$, $s_{\#(t)} = s'_{\#(t)}$. We do a case distinction on whether $reg(a) = \bot$.

- If $reg(a) = \bot$, then a is in $TLoc_t$. Since these actions occur only in $Act_{\#(t)}$, we merely need to show that a is enabled in $s'_{\#(t)}$ to establish that a is enabled in s'. Since a is enabled in s, it is enabled in $s_{\#(t)}$. We have established that $s_{\#(t)} = s'_{\#(t)}$, so a is enabled in s'.
- If $reg(a) \neq \bot$, then a is not a thread local action, and must be a register interface or register local action. In both cases, it is an action in $Act_{\#(r)}$. If it is a register local action, it solely remains to prove that a is enabled in $s'_{\#(r)}$; if it is a register interface action, we must also prove that a is enabled in $s'_{\#(t)}$. The latter follows immediately, in the case that a is a register interface action, from the observation that $s_{\#(t)} = s'_{\#(t)}$ and a being enabled in $s_{\#(t)}$. In both cases, it therefore suffices to prove that a is enabled in $s'_{\#(r)}$. From the construction of $Trans$ it follows that a is enabled in $s_{\#(r)}$. Note that if $reg(b) \neq r$, then $s_{\#(r)} = s'_{\#(r)}$ and so a is trivially enabled in $s'_{\#(r)}$. Therefore, we continue under the assumption that $reg(b) = r$. Since b is the action label of a transition from s to s', and $reg(b) = r \neq \bot$, we know that $(s_{\#(r)}, b, s'_{\#(r)}) \in Trans_{\#(r)}$. We do a further case distinction on what action a is:
  - If $a = sr_{t,r}$ or $a = sw_{t,r}(d)$ for some $d \in \mathbb{D}_r$, then for all three register models, it follows that $t \notin (rds(s_{\#(r)}) \cup wrts(s_{\#(r)}))$. We need to show that $t \notin (rds(s'_{\#(r)}) \cup wrts(s'_{\#(r)}))$ as well. From $thr(b) \neq t$ it follows that $b \neq sr_{t,r}$ and $b \neq sw_{t,r}(d')$ for any $d' \in \mathbb{D}_r$. We know by Observation 13 that if $t \notin rds(s_{\#(r)})$ then $t \notin rds(s'_{\#(r)})$ and that if $t \notin wrts(s_{\#(r)})$ then $t \notin wrts(s'_{\#(r)})$. Therefore, $t \notin (rds(s'_{\#(r)}) \cup wrts(s'_{\#(rid)}))$. Hence, it follows for all three register models that a is enabled in $s'_{\#(r)}$ and thus enabled in s'.
  - If $a = fr_{t,r}(d)$ for some $d \in \mathbb{D}_r$, then to prove that a is enabled in $s'_{\#(r)}$, we do a case distinction on which type of register r is:
    - If r is a safe register, then it follows from a being enabled in $s_{\#(r)}$ that $t \in rds(s_{\#(r)})$ and either $\neg ovrl(s_{\#(r)}, t)$ and $d = stor(s_{\#(r)})$, or $ovrl(s_{\#(r)}, t)$. To show that a is enabled in $s'_{\#(r)}$, we need to show $t \in rds(s'_{\#(r)})$, and $d = stor(s'_{\#(r)})$ or $ovrl(s'_{\#(r)}, t)$. From $thr(b) \neq t$ it follows that $b \neq fr_{t,r}(d')$ for some $d' \in \mathbb{D}_r$. Hence, it follows from Observation 13 that if $t \in rds(s_{\#(r)})$ then $t \in rds(s'_{\#(r)})$. Additionally, if $ovrl(s_{\#(r)}, t)$ then $\neg ovrl(s'_{\#(r)}, t)$ would only be possible if $b = sr_{t,r}$ or $b = sw_{t,r}(d')$ for some $d' \in \mathbb{D}_r$. Since $thr(b) \neq t$, we can conclude that this is not the case and therefore that if $ovrl(s_{\#(r)}, t)$, then $ovrl(s'_{\#(r)}, t)$. Hence, if $ovrl(s_{\#(r)}, t)$ then a is enabled in $s'_{\#(r)}$ and thus a is enabled in s'. We continue under the assumption

that $\neg ovrl(s_{\#(r)}, t)$, and thus that $d = stor(s_{\#(r)})$. Assume towards a contradiction that $\neg ovrl(s'_{\#(r)}, t)$ and $d \neq stor(s'_{\#(r)})$. If $stor(s_{\#(r)}) \neq stor(s'_{\#(r)})$, then b must be $fw_{t',r}$ or $ow_{t',r}$ for some $t' \neq t$. Since there are no order write actions in the safe register model, it must be the case that b is a finish write action. As b is enabled in $s_{\#(r)}$, this means that $t' \in wrts(s_{\#(r)}) \neq \emptyset$. Since $t \in rds(s_{\#(r)})$, it follows from Lemma 14 that $ovrl(s_{\#(r)}, t)$. This contradicts our assumption that $\neg ovrl(s_{\#(r)}, t)$, thus we can conclude that we must have $ovrl(s'_{\#(r)}, t)$ or $d = stor(s'_{\#(r)})$, and consequently a is enabled in $s'_{\#(r)}$. Hence, a is enabled in s'.

- If r is a regular register, then it follows from a being enabled in $s_{\#(r)}$ that $t \in rds(s_{\#(r)})$ and $d \in posv(s_{\#(r)}, t)$. To show a is enabled in $s'_{\#(r)}$, it suffices to show that $t \in rds(s'_{\#(r)})$ and $d \in posv(s'_{\#(r)}.t)$. From Observation 13 and $thr(b) \neq t$, it follows that if $t \in rds(s_{\#(r)})$, then $t \in rds(s'_{\#(r)})$. Additionally, as can be seen in Observation 13, the set $posv$ for a specific thread id only grows over time until it is reset when that thread starts a read. Therefore, it is only possible that $d \in posv(s_{\#(r)}, t)$ and $d \notin posv(s'_{\#(r)}, t)$ are both true if $b = sr_{t,r}$. Since $thr(b) \neq t$, we can conclude that $d \in posv(s'_{\#(r)}, t)$. Thus, a is enabled in $s'_{\#(r)}$ and therefore a is enabled in s'.
- If r is an atomic register, then it follows from a being enabled in $s_{\#(r)}$ that $t \in rds(s_{\#(r)})$, $t \notin pend(s_{\#(r)})$ and $d = rec(s_{\#(r)}, t)$. From Observation 13 it follows that when $thr(b) \neq t$, then $t \in rds(s_{\#(r)})$ implies $t \in rds(s'_{\#(r)})$ and $t \notin pend(s_{\#(r)})$ implies $t \notin pend(s'_{\#(r)})$. Similarly, since $thr(b) \neq t$, $rec(s_{\#(r)}, t) = rec(s'_{\#(r)}, t)$. Thus, a is enabled in $s'_{\#(r)}$ and therefore also in s'.

In all three cases, a is enabled in s'.

- If $a = fw_{t,r}$, then what we need to show about $s'_{\#(r)}$ to establish that a is enabled differs depending on which type of register r is. If r is safe, then it suffices to show $t \in wrts(s'_{\#(r)})$. If r is regular or atomic, we need to show $t \in wrts(s'_{\#(r)})$ and $t \notin pend(s'_{\#(r)})$. From the definitions of these access function as given in Observation 13, and $thr(b) \neq t$, it follows that $t \in wrts(s_{\#(r)})$ implies $t \in wrts(s'_{\#(r)})$ and $t \notin pend(s_{\#(r)})$ implies $t \notin pend(s'_{\#(r)})$. Consequently, if a is enabled in $s_{\#(r)}$ then it is also enabled in $s'_{\#(r)}$. Thus, we can conclude that a is enabled in $s'_{\#(r)}$ and therefore also in s'.

- If $a = ow_{t,r}$, then r could be a regular or atomic register. From the two models, we know that since a is enabled in $s_{\#(r)}$, we have $t \in wrts(s_{\#(r)})$ and $t \in pend(s_{\#(r)})$. For both models, to show a is enabled in $s'_{\#(r)}$, it suffices to show $t \in wrts(s'_{\#(r)})$ and $t \in pend(s'_{\#(r)})$. From Observation 13 and $thr(b) \neq t$, it follows that if $t \in wrts(s_{\#(r)})$ then $t \in wrts(s'_{\#(r)})$ and if $t \in pend(s_{\#(r)})$ then $t \in pend(s'_{\#(r)})$. Thus, a is enabled in $s'_{\#(r)}$ and therefore a is enabled in s'.

- If $a = or_{t,r}$ then r must be an atomic register. From the atomic register model and a being enabled in $s_{\#(r)}$, we know that $t \in rds(s_{\#(r)})$ and $t \in pend(s_{\#(r)})$. From the definitions of the access functions as given in Observation 13 and $thr(b) \neq t$, it follows that $t \in rds(s_{\#(r)})$ implies $t \in rds(s'_{\#(r)})$ and $t \in pend(s_{\#(r)})$ implies $t \in pend(s'_{\#(r)})$. Thus, when a is enabled in $s_{\#(r)}$ it is also enabled in $s'_{\#(r)}$. We conclude that a is enabled in s'.

We have shown in all cases that a is enabled in s'. Hence, our thread-register models are thread consistent with respect to the mapping thr. ◀

D The tread interference relation is a concurrency relation

▶ **Lemma 8.** $\smile\!\!\bullet_T$ is a concurrency relation for M.

Proof. To prove $\smile\!\!\bullet_T$ is a concurrency relation, we need to prove the two properties of concurrency relations.

1. It follows directly from the definition of $\smile\!\!\bullet_T$ that it is irreflexive: trivially $thr(a) = thr(a)$ for all $a \in Act$.
2. Let a be an arbitrary action in Act and let s be an arbitrary state in $\mathcal{S}$. Assume that a is enabled in s and that there exists a path π from s to some s' such that $a \smile\!\!\bullet_T b$ for all b occurring on π. We prove that a is enabled in s'. We do induction on the length of π.
   - For the base case, $|\pi| = 0$, we observe that if π has length 0, then $s = s'$ and hence trivially a is enabled in s'.
   - Let $i \geq 0$ and assume that the claim holds for all paths from s of length i; we prove that it also holds for all paths from s of length $i+1$. Let π be a path from s to some state $s' \in \mathcal{S}$ with $|\pi| = i+1$ such that $a \smile\!\!\bullet_T b$ for all b occurring on π. Then there exists a path π' from s to some state s'' of length i such that $a \smile\!\!\bullet_T b$ for all b occurring on π', and $\pi = \pi' c s'$ for some $c \in Act$. Note that by assumption on π, $a \smile\!\!\bullet_T c$. By the induction hypothesis, a is enabled in s''. Since $a \smile\!\!\bullet_T c$, it follows by definition of $\smile\!\!\bullet_T$ that $thr(a) \neq thr(c)$ and thus, since M is thread-consistent by Lemma 6, it follows that a is enabled in s'.

   We conclude that the property holds for all paths π. ◀

E An explicit characterisation of complete paths

For a given mutual exclusion algorithm, let M be the LTS of its thread-register model from Section 4, using one of the register models employed in this paper—cf. Appendix B. Recalling that M is a parallel composition of thread and register processes, each state in M is a tuple $s = (s_1, \ldots, s_k)$, where each of the indices $i \in \{1, \ldots, k\}$ corresponds to a thread $t \in \mathbb{T}$ or a register $r \in \mathbb{R}$; as in the proof of Lemma 6 in Appendix C we denote the corresponding component s_i as $s_{\#(t)}$ or $s_{\#(r)}$; it is a state in the LTS T_t or R_r.

▶ **Lemma 15.** If s is a reachable state of M such that $s_{\#(r)}$ enables an action $ow_{t,r}$ or $or_{t,r}$ in R_r, then $s_{\#(t)}$ enables an action $fw_{t,r}$ or $fr_{t,r}(d)$, respectively, in T_t. Moreover, if $s_{\#(r)}$ enables $c = fw_{t,r}$ or $c = fr_{t,r}(d)$, then also $s_{\#(t)}$ enables c, and thus s enables c in M.

Proof. We will provide the proof for the cases that $s_{\#(r)}$ enables an action $ow_{t,r}$ or $fw_{t,r}$. The case for read actions proceeds along the exact same lines.

Given a register $r \in \mathbb{R}$ and a thread $t \in \mathbb{T}$, in any execution path for any of the register models from Figures 2–4, the actions c with $thr(c) = t$ and $reg(c) = r$ occur strictly in the order $sw_{t,r}(d)$ – $ow_{t,r}$ – $fw_{t,r}$. This is a direct consequence of the conditions $t \in wrts(s)$ and $t \in pend(s)$ in Figures 2–4. Moreover, the actions $sw_{t,r}(d)$ for $d \in \mathbb{D}_r$ and $fw_{t,r}$ can occur only in synchronisation between the register r and the thread t. Thus, if s is a reachable state of M such that $s_{\#(r)}$ enables an action $ow_{t,r}$ or $fw_{t,r}$, then the last synchronisation between r and t must have been an action $sw_{t,r}(d)$.

In Section 4 we postulated that on all paths from the initial state of T_t, each transition labelled $sw_{t,r}(d)$ for some $r \in \mathbb{R}$ and $d \in \mathbb{D}_r$ must go to a state where only the action $fw_{t,r}$ is enabled. Since the last synchronisation between r and t was on an action $sw_{t,r}(d)$, this $fw_{t,r}$ cannot have occurred yet. Hence $s_{\#(t)}$ enables the action $fw_{t,r}$.

In case $s_{\#(r)}$ enables $fw_{t,r}$, then (by the above) both $s_{\#(r)}$ and $s_{\#(t)}$ enable this action, and therefore it is also enabled by s. ◀

▶ **Lemma 16.** If s is a state of M such that $s_{\#(t)}$ enables an action $fw_{t,r}$ or $fr_{t,r}(d)$, then either $ow_{t,r}$ or $or_{t,r}$ or $fw_{t,r}$ or $fr_{t,r}(d')$ for some $d' \in \mathbb{D}_r$ is enabled by $s_{\#(r)}$.

Proof. In Section 4 we postulated that in T_t transitions labelled $fr_{t,r}(d)$ are only enabled right after performing a transition labelled $sr_{t,r}$. As in M the $sr_{t,r}$-transition must have been a synchronisation between thread t and register r, and the register cannot execute an action $fr_{t,r}(d)$ without synchronising with t, in each of our register models we have $t \in rds(s)$. Consequently, by Figures 2–4, either $or_{t,r}$ or $fr_{t,r}(d')$ for some $d' \in \mathbb{D}_r$ is enabled by $s_{\#(r)}$.

The proof for write actions proceeds along the same lines. ◀

▶ **Definition 17.** Given any path π starting in the initial state of M, and given any thread $t \in \mathbb{T}$, if π has a suffix π' on which no actions b with $thr(b) = t$ occur, starting from a state s of π, then $s'_{\#(t)} = s_{\#(t)}$ for all states s' in π' and we define $end_t(\pi) = s_{\#(t)}$.

Call action $a \in Act$ *thread-enabled* by π if, for $t = thr(a)$, π contains only finitely many actions b with $thr(b) = t$ and a is enabled in the state $end_t(\pi)$ of the LTS T_t.

Recall from Definition 7 that $a \not\smile^{\bullet}_T b$ iff $thr(a) = thr(b)$. Hence a path π is $\mathcal{B}$-$\smile^{\bullet}_T$-unjust if, and only if, it has a suffix π' such that an action $a \in \overline{\mathcal{B}}$ is enabled in the initial state of π', but π does not contain any action b with $thr(a) = thr(b)$. Using this, we obtain the following characterisation of the $\mathcal{B}$-$\smile^{\bullet}_T$-just paths of M.

▶ **Proposition 18.** A path π starting in the initial state of M, is $\mathcal{B}$-$\smile^{\bullet}_T$-just if, and only if, π thread-enables no actions $a \in \overline{\mathcal{B}}$.

Proof. Suppose π thread-enables an action $a \in \overline{\mathcal{B}}$. We have to show that π is not just. Let $t = thr(a)$ and $r = reg(a)$. Let π' be a suffix of π on which no actions b with $thr(b) = t$ occur. Let s be the initial state of π'. So $s_{\#(t)} = end_t(\pi)$ and $s_{\#(t)}$ enables a.

In case $r = \bot$, as $s_{\#(t)}$ enables a and a does not require synchronisation with any register, also s enables a. As π' does not contain actions b with $a \not\smile^{\bullet}_T b$, the path π is not just.[6]

So assume that $r \in \mathbb{R}$. Based on Figures 2–4, any state s'_r of r enables either (i) both $sr_{t,r}$ and $sw_{t,r}(d)$ for all $d \in \mathbb{D}_r$, (ii) $or_{t,r}$ or $ow_{t,r}$, or (iii) $fw_{t,r}$ or $fr_{t,r}(d)$ for some $d \in \mathbb{D}_r$.

If state $s_{\#(r)}$ in R_r enables $c = or_{t,r}$ or $c = ow_{t,r}$, also s enables c, since c does not require synchronisation with thread t. As π' contains no actions b with $c \not\smile^{\bullet}_T b$, using that $thr(c) = t$, the path π is not just.[7]

In case state $s_{\#(r)}$ in R_r enables an action $c = fr_{t,r}(d)$ or $c = fw_{t,r}$, by Lemma 15 above also state s enables c. As π' contains no actions b with $c \not\smile^{\bullet}_T b$, the path π is not just.[7]

We may now restrict attention to case (i) above, that $s_{\#(r)}$ enables both $sr_{t,r}$ and $sw_{t,r}(d)$ for all $d \in \mathbb{D}_r$. In this setting we proceed with a case distinction on the action a, which must be of the form $sr_{t,r}$, $sw_{t,r}(d)$, $fr_{t,r}(d)$ or $fw_{t,r}$, since it is an action in LTS T_t and $r \neq \bot$. By Lemma 16, the case that $a = fw_{t,r}$ or $a = fr_{t,r}(d)$ for some $d \in \mathbb{D}_r$ is already subsumed by the cases considered above. Thus we may restrict attention to the case that $a = sr_{t,r}$ or $a = sw_{t,r}(d)$ for some $d \in \mathbb{D}_r$. In this case also s enables a. As π' does not contain actions b with $a \not\smile^{\bullet}_T b$, the path π is not just.[8]

---

[6] When rereading this proof as part of the proof of Proposition 19 we also use that $\forall r \in \mathbb{R}.\ a \notin start(r)$.
[7] When rereading this proof as part of the proof of Proposition 19 we also use that $c \notin start(r)$.
[8] When reusing this in the proof of Proposition 19, we also use that π' contains no actions $b \in start(r)$.

For the other direction, suppose that π is $\mathcal{B}$-$\smile\!\!\bullet_T$-just. We have to establish that π thread-enables an action $a \in \overline{\mathcal{B}}$. Let π' be a suffix of π such that an action $a \in \overline{\mathcal{B}}$ is enabled in the initial state s of π', but π' contains no action b such that $a \not\smile\!\!\bullet_T b$. Let $t = \mathit{thr}(a)$. Then π' contains no action b with $\mathit{thr}(b) = t$. In case $a = \mathit{ow}_{t,r}$ or $a = \mathit{or}_{t,r}$, then by Lemma 15 above $\mathit{end}_t(\pi) = s_{\#(t)}$ enables an action $c = \mathit{fw}_{t,r}$ or $c = \mathit{fr}_{t,r}(d)$, and thus π thread-enables the action $c \in \overline{\mathcal{B}}$. Otherwise, $\mathit{end}_t(\pi) = s_{\#(t)}$ enables a and thus π thread-enables $a \in \overline{\mathcal{B}}$. ◀

Interestingly, this explicit characterisation of the $\mathcal{B}$-$\smile\!\!\bullet_T$-just paths in our thread-register models is independent on whether we employ safe, regular or atomic registers.

Next, we provide similar characterisations of the $\mathcal{B}$-$\smile\!\!\bullet_C$-just paths, for $C \in \{A, I, S\}$. For a given register $r \in \mathbb{R}$ we use the abbreviation $\mathit{start}(r)$ for the set of all actions $\mathit{sr}_{t,r}$ and $\mathit{sw}_{t,r}(d)$ for some $t \in \mathbb{T}$ and $d \in \mathbb{D}_r$. Recall from Definitions 7–11 that $a \not\smile\!\!\bullet_A b$ iff either $\mathit{thr}(a) = \mathit{thr}(b)$ or $a, b \in \mathit{start}(r)$ for some $r \in \mathbb{R}$. Hence a path π is $\mathcal{B}$-$\smile\!\!\bullet_A$-unjust if, and only if, it has a suffix π' such that an action $a \in \overline{\mathcal{B}}$ is enabled in the initial state of π', but π does not contain any action b such that $\mathit{thr}(a) = \mathit{thr}(b)$ or $a, b \in \mathit{start}(r)$ for some $r \in \mathbb{R}$.

▶ **Proposition 19.** A path π starting in the initial state of M, is $\mathcal{B}$-$\smile\!\!\bullet_A$-just if, and only if,

a) π thread-enables no actions $a \in \overline{\mathcal{B}}$ other than actions from $\mathit{start}(r)$ for some $r \in \mathbb{R}$, and

b) if, for some $r \in \mathbb{R}$, an action $a \in \mathit{start}(r)$ is thread-enabled by π, then π contains infinitely many occurrences of actions $b \in \mathit{start}(r)$.

Proof. Suppose π thread-enables an action $a \in \overline{\mathcal{B}}$, say with $t = \mathit{thr}(a)$ and $r = \mathit{reg}(a)$, such that if $a \in \mathit{start}(r)$ then π contains only finitely many actions $b \in \mathit{start}(r)$. We have to show that π is not $\mathcal{B}$-$\smile\!\!\bullet_A$-just. Let π' be a suffix of π in which no actions b with $\mathit{thr}(b) = t$ occur; in case $a \in \mathit{start}(r)$ we moreover choose π' such that it contains no actions $b \in \mathit{start}(r)$. From here on, the proof proceeds exactly as the one of Proposition 18, but reading $\not\smile\!\!\bullet_A$ for $\not\smile\!\!\bullet_T$.

For the other direction, suppose that π is not $\mathcal{B}$-$\smile\!\!\bullet_A$-just. We have to establish that (a) π thread-enables an action $a \in \overline{\mathcal{B}}$, and (b) in case $a \in \mathit{start}(r)$ then π contains only finitely many actions $b \in \mathit{start}(r)$. Let π' be a suffix of π such that an action $a \in \overline{\mathcal{B}}$ is enabled in the initial state s of π', but π' contains no action b such that $a \not\smile\!\!\bullet_A b$, that is, π' contains no action b with $\mathit{thr}(a) = \mathit{thr}(b)$ or $a, b \in \mathit{start}(r)$ for some $r \in \mathbb{R}$. The proof of (a) above proceeds as in the proof of Proposition 18, and (b) is now a trivial corollary. ◀

Recall from Definitions 7–10 that $a \not\smile\!\!\bullet_I b$ iff either $\mathit{thr}(a) = \mathit{thr}(b)$ or $a, b \in \mathit{start}(r)$ for some $r \in \mathbb{R}$, with $\mathit{sw}?(a) \vee \mathit{sw}?(b)$. Hence a path π is $\mathcal{B}$-$\smile\!\!\bullet_I$-unjust if, and only if, it has a suffix π' such that an action $a \in \overline{\mathcal{B}}$ is enabled in the initial state of π', but π does not contain any action b such that $\mathit{thr}(a) = \mathit{thr}(b)$ or $a, b \in \mathit{start}(r)$ for some $r \in \mathbb{R}$, with $\mathit{sw}?(a) \vee \mathit{sw}?(b)$.

Similarly, $a \not\smile\!\!\bullet_S b$ iff either $\mathit{thr}(a) = \mathit{thr}(b)$ or $a, b \in \mathit{start}(r)$ for some $r \in \mathbb{R}$, with $\mathit{sw}?(b)$. Hence a path π is $\mathcal{B}$-$\smile\!\!\bullet_S$-unjust if, and only if, it has a suffix π' such that an action $a \in \overline{\mathcal{B}}$ is enabled in the initial state of π', but π does not contain any action b such that $\mathit{thr}(a) = \mathit{thr}(b)$ or $a, b \in \mathit{start}(r)$ for some $r \in \mathbb{R}$, with $\mathit{sw}?(b)$.

▶ **Proposition 20.** A path π starting in the initial state of M, is $\mathcal{B}$-$\smile\!\!\bullet_I$-just if, and only if,

a) π thread-enables no actions $a \in \overline{\mathcal{B}}$ other than actions from $\mathit{start}(r)$ for some $r \in \mathbb{R}$,

b) if an action $\mathit{sw}_{t,r}(d)$ is thread-enabled by π, then π contains infinitely many occurrences of actions $b \in \mathit{start}(r)$, and

c) if an action $\mathit{sr}_{t,r}$ is thread-enabled by π, then π contains infinitely many occurrences of actions b of the form $\mathit{sw}_{t',r}(d)$ for some $t' \in \mathbb{T}$ and $d' \in \mathbb{D}_r$.

The proof of this proposition, and the next one, proceeds just like the one of Proposition 19.

▶ **Proposition 21.** A path π starting in the initial state of M, is $\mathcal{B}$-$\smile\!\!\bullet_S$-just if, and only if,

a) π thread-enables no actions $a \in \overline{\mathcal{B}}$ other than actions from $start(r)$ for some $r \in \mathbb{R}$,

b) if an action $a \in start(r)$ is thread-enabled by π, then π contains infinitely many occurrences of actions b of the form $sw_{t',r}(d)$ for some $t' \in \mathbb{T}$ and $d' \in \mathbb{D}_r$.

An interesting consequence of these propositions is that whether or not a given path π is $\mathcal{B}$-$\smile\!\!\bullet_C$-just, for some $C \in \{T, S, I, A\}$, is completely determined by the set of actions that are thread-enabled by π, and by the function $occ_\pi : Act \to \mathbb{N} \cup \{\infty\}$ that tells for each action how often it occurs in π.

F Register models that avoid overlap

To accurately capture the memory model of blocking reads and writes, we need not only the concurrency relation $\smile\!\!\bullet_A$, but also a register model that does not allow any read or write to start when another read or write to the same register is in progress. This can be achieved by changing, in Figure 4, the condition $t \notin (rds(s) \cup wrts(s))$, occurring in the first two lines, into $(rds(s) \cup wrts(s)) = \emptyset$.

Similarly, in the blocking model with concurrent reads, where reads and writes have to await the completion of in-progress writes, but only writes have to await the completion of in-progress reads, the first two lines of Figure 4 become

$$\begin{aligned} &(t \notin rds(s) \wedge wrts(s) = \emptyset) \to sr_{t,r} \cdot Reg_{ato}(r, usr(s,t)) \\ + \quad &((rds(s) \cup wrts(s)) = \emptyset) \to sw_{t,r}(d) \cdot Reg_{ato}(r, usw(s,t,d)) \end{aligned}$$

To capture the model with blocking writes and non-blocking reads, we make the same alterations as above, except that writes do not need to wait for in-progress reads. Hence, the first two lines become

$$\begin{aligned} &(t \notin rds(s) \wedge wrts(s) = \emptyset) \to sr_{t,r} \cdot Reg_{ato}(r, usr(s,t)) \\ + \quad &(wrts(s) = \emptyset) \to sw_{t,r}(d) \cdot Reg_{ato}(r, usw(s,t,d)) \end{aligned}$$

Note that this only models the blocking behaviour described in Section 1; we do not model that a write causes a read to be aborted and subsequently resumed.

For our verifications we have not made these modifications, but simply reused the register model of Figure 4. Below, we explain why this leads to the same verification results.

All our correctness properties for mutual exclusion protocols are linear-time properties: they hold for a process, modelled as an LTS, iff they hold for all complete paths starting in the initial state of that LTS. Moreover, whether they hold for a particular path depends solely on the labels of the transitions in that path, and in fact only on those transitions labelled c_t or nc_t, for some $t \in \mathbb{T}$. This is witnessed by the modal μ-calculus formulae for these properties given in Appendix H. For the (sole) purpose of deciding whether mutual exclusion, deadlock freedom or starvation freedom hold for a given LTS, once its is determined which paths are complete, all other actions may be considered internal, or hidden.

▶ **Definition 22.** Given a path $\pi = s_0 a_1 s_1 a_2 s_2 \ldots$, let $\ell(\pi) = a_1 a_2 \ldots$ be the sequence of actions occurring on that path, and let $\ell^-(\pi)$ be the result of omitting from $\ell(\pi)$ all actions other than c_t or nc_t, for some $t \in \mathbb{T}$.

Given a completeness criterion C, the set of *weak C-complete traces* $WCT_C(P)$ of an LTS P consists of those finite and infinite strings $\ell^-(\pi)$ for π a C-complete path starting in the initial state of P. Two LTSs P and Q are *weak completed trace equivalent* w.r.t. C, notation $P =^C_{WCT} Q$ if, and only if, $WCT_C(P) = WCT_C(Q)$.

It now follows that, given a completeness criterion C, two weak completed trace equivalent LTSs satisfy the same correctness properties for mutual exclusion protocols.

For a given mutual exclusion algorithm, let M be the LTS of its thread-register model, using the atomic register model from Figure 4. Moreover, let M_A be the variant of M that employs the above register model for blocking reads and writes; let M_I be the variant for the blocking model with concurrent reads, and let M_S be the one for the model of blocking writes and non-blocking reads.

Now, using A, I and S as abbreviations for the completeness criteria $JA_{\mathcal{B}}^{\smile\!\bullet A}$, $JA_{\mathcal{B}}^{\smile\!\bullet I}$ and $JA_{\mathcal{B}}^{\smile\!\bullet S}$, we will show that $M_A =^{A}_{WCT} M$, $M_I =^{I}_{WCT} M$ and $M_S =^{S}_{WCT} M$. This implies that it makes no difference whether, for the verifications based on the (partly) blocking memory models, we use the register models fine-tuned for that memory model as described above, or the model for atomic registers from Figure 4.

In the subsequent argument, let $C \in \{A, I, S\}$.

First, we must establish that $\smile\!\bullet_C$ is a valid concurrency relation for M_C. That the relations are irreflexive solely depends on the actions of an LTS, and hence still clearly holds within the context of M_A, M_I and M_S. It is less immediately obvious that the second property of concurrency relations still holds, particularly because the fine-tuned models are not thread-consistent: it is for example possible for $sr_{t,r}$ to be disabled by the occurrence of $sw_{t',r}(d)$ in all three models, even when $t \neq t'$. However, in all three cases, if an action a is disabled by the occurrence of an action b, then $a \not\smile\!\bullet_C b$:

- In M_A, $sr_{t,r}$ and $sw_{t,r}(d)$ can be disabled by the occurrence of $sr_{t',r}$ or $sw_{t',r}(d')$. Accordingly, in $\smile\!\bullet_A$ start read actions and start write actions both interfere with both start read and start write actions on the same register.
- In M_I, $sr_{t,r}$ can be disabled by the occurrence of $sw_{t',r}(d)$. Accordingly, $\smile\!\bullet_I$ allows start write actions to interfere with start read actions on the same register. Additionally, $sw_{t,r}(d)$ can be disabled by the occurrence of $sr_{t',r}$ or $sw_{t',r}(d')$. Indeed, $\smile\!\bullet_I$ allows start reads and start writes to interfere with start writes on the same register.
- In M_S, $sr_{t,r}$ and $sw_{t,r}(d)$ can both be disabled by an occurrence of $sw_{t',r}(d')$. Accordingly, $\smile\!\bullet_S$ allows start writes to interfere with both start reads and start writes on the same register.

Thus, in all three register models an action can only be disabled by the occurrence of an action that interferes with it according to the appropriate concurrency relation. Consequently, if there is a transition $s \xrightarrow{b} s'$ such that an action a is enabled in s and $a \smile\!\bullet_C b$, then a is enabled in s'. By induction, similarly to the proof of Lemma 8, the second property of concurrency relations is satisfied, and we can apply these concurrency relations to their respective models.

Next, we will show that the most relevant results of Appendix E apply also to the register models introduced above. First observe that Lemmas 15 and 16 apply equally well, with the same proofs, when reading M_A, M_I or M_S for M, since the only difference between these models is when start actions are enabled, and there lemmas refer only to order and finish actions.

▶ **Lemma 23.** Let T_r be the LTS of any register r, either defined as in one of the Figures 2–4, or using one of three register models defined at the beginning of this appendix. Then any state s_r of r enables either (i) $sr_{t,r}$ and $sw_{t,r}(d)$ for all $t \in \mathbb{T}$ and all $d \in \mathbb{D}_r$, or (ii) $or_{t,r}$ or $ow_{t,r}$ for some $t \in \mathbb{T}$, or (iii) $fw_{t,r}$ or $fr_{t,r}(d)$ for some $t \in \mathbb{T}$ and $d \in \mathbb{D}_r$.

Proof. In case $rds(s_r) \cup wrts(s_r) = \emptyset$, option (i) applies. This can be seen from Figures 2–4, and from the description of three register models defined at the beginning of this appendix.

Likewise, in case $rds(s_r) \cup wrts(s_r) \neq \emptyset$, one of (ii) or (iii) must apply. (In the case of regular registers, we use the easily checked invariant that if $t \in rds(s)$ then $posv(s,t) \neq \emptyset$.) ◀

▶ **Lemma 24.** Let T_r be the LTS of any register r, as defined at the beginning of this appendix for the register model with blocking writes and non-blocking reads, i.e. M_S. Then any state s_r of r enables either (i) $sw_{t,r}(d)$ for all $t \in \mathbb{T}$ and all $d \in \mathbb{D}_r$, or (ii) $ow_{t,r}$ for some $t \in \mathbb{T}$, or (iii) $fw_{t,r}$ for some $t \in \mathbb{T}$. If case (i) applies then, for each $t' \in \mathbb{T}$, s_r enables either (a) $sr_{t',r}$, or (b) $or_{t',r}$, or (c) $fr_{t',r}(d)$ for all $d \in \mathbb{D}_r$.

Proof. In case $wrts(s_r) = \emptyset$, option (i) applies. This can be seen from the description of this register model. Likewise, in case $wrts(s_r) \neq \emptyset$, one of (ii) or (iii) must apply.

Assuming $wrts(s_r) = \emptyset$, if $t \notin rds(s_r)$, option (a) applies and otherwise (b) or (c) apply. ◀

▶ **Proposition 25.** The characterisations of the $\mathcal{B}$-$\smile\!\!\bullet_C$-just paths of M from Propositions 19–21 apply equally well to M_C.

Proof. Below we give a proof for the case $C = A$, most of which applies to all three cases $C \in \{A, I, S\}$. The parts that are specific to the case $C = A$ are coloured orange, and afterwards we will give the case $C = S$ with the differences between it and $C = A$ similarly highlighted. The case $C = I$ is treated at the end.

We must prove that a path π starting in the initial state of M_A, is $\mathcal{B}$-$\smile\!\!\bullet_A$-just if, and only if, a) π thread-enables no actions $a \in \overline{\mathcal{B}}$ other than actions from $start(r)$ for some $r \in \mathbb{R}$, and b) if, for some $r \in \mathbb{R}$, an action $a \in start(r)$ is thread-enabled by π, then π contains infinitely many occurrences of actions $b \in start(r)$.

Suppose π thread-enables an action $a \in \overline{\mathcal{B}}$, say with $t = thr(a)$ and $r = reg(a)$, such that if $a \in start(r)$ then π contains only finitely many actions $b \in start(r)$. In the latter case π contains only finitely many actions b with $reg(b) = r$, because actions c with $thr(c) = t'$ and $reg(c) = r$ must occur strictly in the order $sw_{t',r}(d) - ow_{t',r} - fw_{t',r}$.

We have to show that π is not $\mathcal{B}$-$\smile\!\!\bullet_C$-just. Let π' be a suffix of π in which no actions b with $thr(b) = t$ occur; in case $a \in start(r)$ we moreover choose π' such that it contains no actions b with $reg(b) = r$. Let s be the initial state of π'. So $s_{\#(t)} = end_t(\pi)$ enables a.

In case $r = \bot$, as $s_{\#(t)}$ enables a and a does not require synchronisation with any register, also s enables a. As π' does not contain actions b with $a \not\smile\!\!\bullet_C b$, the path π is not just.

So assume that $r \in \mathbb{R}$. We proceed with a case distinction on the action a, which must be of the form $sr_{t,r}$, $sw_{t,r}(d)$, $fr_{t,r}(d)$ or $fw_{t,r}$, since it is an action in LTS T_t and $r \neq \bot$.

First assume that $a = fw_{t,r}$ or $a = fr_{t,r}(d)$ for some $d \in \mathbb{D}_r$. By Lemma 16, either $ow_{t,r}$ or $or_{t,r}$ or $fw_{t,r}$ or $fr_{t,r}(d')$ for some $d' \in \mathbb{D}_r$ is enabled by state $s_{\#(r)}$ in R_r. In case $s_{\#(r)}$ enables $c = or_{t,r}$ or $c = ow_{t,r}$, also s enables c, since c does not require synchronisation with thread t. As π' contains no actions b with $c \not\smile\!\!\bullet_C b$, using that $thr(c) = t$, the path π is not just. In case $s_{\#(r)}$ enables an action $c = fr_{t,r}(d)$ or $c = fw_{t,r}$, by Lemma 15 also state s enables c. As π' contains no actions b with $c \not\smile\!\!\bullet_C b$, the path π is not just.

Thus we may restrict attention to the case that $a = sr_{t,r}$ or $a = sw_{t,r}(d)$ for some $d \in \mathbb{D}_r$, that is, $a \in start(r)$. So henceforth we may use that π' contains no actions b with $reg(b) = r$.

We proceed by making a case distinction on the three possibilities described by Lemma 23 for actions enabled by $s_{\#(r)}$. In case (i) of Lemma 23, $s_{\#(r)}$ enables $sr_{t',r}$ and $sw_{t',r}(d')$ for all $t' \in \mathbb{T}$ and all $d' \in \mathbb{D}_r$, and thus in particular the action a. Consequently, also s enables a. As π' does not contain actions b with $thr(b) = t$ or actions $b \in start(r)$, it contains no action b with $a \not\smile\!\!\bullet_C b$. Hence, the path π is not just.

In case (ii) $s_{\#(r)}$, and hence also s, enables $or_{t',r}$ or $ow_{t',r}$ for some $t' \in \mathbb{T}$. Thus, by Lemma 15, $s_{\#(t')}$ enables an action $fw_{t',r}$ or $fr_{t',r}(d)$. By the assumptions of Section 4, this

implies that $s_{\#(t')}$ only enables actions b with $reg(b) = r$. Since π' does not contain such b, it cannot contain any action c with $thr(c) = t'$ either. Consequently, π is not just.

In case (iii) $s_{\#(r)}$ enables an action $c = fw_{t',r}$ or $c = fr_{t',r}(d)$ for some $t' \in \mathbb{T}$ and $d \in \mathbb{D}_r$. Thus, by Lemma 15, c is enabled by $s_{\#(t')}$ as well as by s. The rest of the argument proceeds just as for case (ii) above.

For the other direction, note that M_C has the same states as M and can be obtained from M by leaving out some transitions. This implies that any path π in M_C is also a path in M, and moreover, if π is $\mathcal{B}$-$\rightsquigarrow\!\!\!\!\bullet_C$-unjust in M_C then it certainly is $\mathcal{B}$-$\rightsquigarrow\!\!\!\!\bullet_C$-unjust in M. Using this, the reverse direction of this proof is direct consequence of Propositions 19–21.

In the case $C = S$, we must prove that a path π starting in the initial state of M_S, is $\mathcal{B}$-$\rightsquigarrow\!\!\!\!\bullet_S$-just if, and only if, a) π thread-enables no actions $a \in \overline{\mathcal{B}}$ other than actions from $start(r)$ for some $r \in \mathbb{R}$, and b) if an action $a \in start(r)$ is thread-enabled by π, then π contains infinitely many occurrences of actions b of the form $sw_{t',r}(d)$ for some $t' \in \mathbb{T}$ and $d' \in \mathbb{D}_r$. The proof is identical to the $C = A$ case, save for the orange phrases.
Suppose π thread-enables an action $a \in \overline{\mathcal{B}}$, say with $t = thr(a)$ and $r = reg(a)$, such that if $a \in start(r)$ then π contains only finitely many actions b of the form $sw_{t',r}(d)$ for some $t' \in \mathbb{T}$ and $d' \in \mathbb{D}_r$. In the latter case π also contains only finitely many actions b of the form $ow_{t',r}$ or $fw_{t',r}$, because actions c with $thr(c) = t'$ and $reg(c) = r$ must occur strictly in the order $sw_{t',r}(d) - ow_{t',r} - fw_{t',r}$.

We have to show that π is not $\mathcal{B}$-$\rightsquigarrow\!\!\!\!\bullet_C$-just. Let π' be a suffix of π in which no actions b with $thr(b) = t$ occur; in case $a \in start(r)$ we moreover choose π' such that it contains no actions b of the form $sw_{t',r}(d)$, $ow_{t',r}$ or $fw_{t',r}$ for some $t' \in \mathbb{T}$ and $d' \in \mathbb{D}_r$. Let s be the initial state of π'. So $s_{\#(t)} = end_t(\pi)$ enables a.

In case $r = \bot$, as $s_{\#(t)}$ enables a and a does not require synchronisation with any register, also s enables a. As π' does not contain actions b with $a \not\rightsquigarrow\!\!\!\!\bullet_C b$, the path π is not just.

So assume that $r \in \mathbb{R}$. We proceed with a case distinction on the action a, which must be of the form $sr_{t,r}$, $sw_{t,r}(d)$, $fr_{t,r}(d)$ or $fw_{t,r}$, since it is an action in LTS T_t and $r \neq \bot$.

First assume that $a = fw_{t,r}$ or $a = fr_{t,r}(d)$ for some $d \in \mathbb{D}_r$. By Lemma 16, either $ow_{t,r}$ or $or_{t,r}$ or $fw_{t,r}$ or $fr_{t,r}(d')$ for some $d' \in \mathbb{D}_r$ is enabled by state $s_{\#(r)}$ in R_r. In case $s_{\#(r)}$ enables $c = or_{t,r}$ or $c = ow_{t,r}$, also s enables c, since c does not require synchronisation with thread t. As π' contains no actions b with $c \not\rightsquigarrow\!\!\!\!\bullet_C b$, using that $thr(c) = t$, the path π is not just. In case $s_{\#(r)}$ enables an action $c = fr_{t,r}(d)$ or $c = fw_{t,r}$, by Lemma 15 also state s enables c. As π' contains no actions b with $c \not\rightsquigarrow\!\!\!\!\bullet_C b$, the path π is not just.

Thus we may restrict attention to the case that $a = sr_{t,r}$ or $a = sw_{t,r}(d)$ for some $d \in \mathbb{D}_r$, that is, $a \in start(r)$. So henceforth we may use that π' contains no actions b of the form $sw_{t',r}(d)$, $ow_{t',r}$ or $fw_{t',r}$ for some $t' \in \mathbb{T}$ and $d' \in \mathbb{D}_r$.

We proceed by making a case distinction on the three possibilities described by Lemma 24 for actions enabled by $s_{\#(r)}$. In case (i) of Lemma 24, we make a further case distinction, depending on whether $a = sr_{t,r}$ or $a = sw_{t,r}(d)$. In the latter case, $s_{\#(r)}$ enables $sw_{t',r}(d')$ for all $t' \in \mathbb{T}$ and all $d' \in \mathbb{D}_r$, and thus in particular the action a. In the former case, options (b) and (c) for $t' = t$ of Lemma 24 are ruled out, because in those cases Lemma 15 would imply that an action $fr_{t,r}(d)$ would be enabled by $s_{\#}(t)$, but by assumption (see Section 4) this cannot happen in a state enabling $sr_{t,r}$. Thus (a) applies, and also $s_{\#(r)}$ enables action a. Consequently, also s enables a. As π' does not contain actions b with $thr(b) = t$ or actions $b \in start(r)$ with $sw?(b)$, it contains no action b with $a \not\rightsquigarrow\!\!\!\!\bullet_C b$. Hence, the path π is not just.

In case (ii) $s_{\#(r)}$, and hence also s, enables $ow_{t',r}$ for some $t' \in \mathbb{T}$. Thus, by Lemma 15, $s_{\#(t')}$ enables an action $fw_{t',r}$. By the assumptions of Section 4, this implies that $s_{\#(t')}$ only enables the action $b = fw_{t',r}$. Since π' does not contain such b, it cannot contain any action

c with $thr(c) = t'$ either. Consequently, π is not just.

In case (iii) $s_{\#(r)}$ enables an action $c = fw_{t',r}$ for some $t' \in \mathbb{T}$. Thus, by Lemma 15, c is enabled by $s_{\#(t')}$ as well as by s. The rest of the argument proceeds just as for case (ii) above.

For the other direction, note that M_C has the same states as M and can be obtained from M by leaving out some transitions. This implies that any path π in M_C is also a path in M, and moreover, if π is $\mathcal{B}$-$\smile\!\!\bullet_C$-unjust in M_C then it certainly is $\mathcal{B}$-$\smile\!\!\bullet_C$-unjust in M. Using this, the reverse direction of this proof is direct consequence of Propositions 19–21.

Finally, for the case $C = I$ we must prove that a path π starting in the initial state of M_I is $\mathcal{B}$-$\smile\!\!\bullet_I$-just if, and only if, a) π thread-enables no actions $a \in \overline{\mathcal{B}}$ other than actions from $start(r)$ for some $r \in \mathbb{R}$, and b) if an action $sw_{t,r}(d)$ is thread-enabled by π, then π contains infinitely many occurrences of actions $b \in start(r)$, and c) if an action $sr_{t,r}$ is thread-enabled by π, then π contains infinitely many occurrences of actions b of the form $sw_{t',r}(d)$. Let π be a path in M_I that thread-enables an action $a \in \overline{\mathcal{B}}$ such that one of these three conditions is violated. In the case that the violated condition is a) or b), the proof proceeds just as in the case $C = A$; in the case it is condition c) the proof proceeds just as in the case $C = S$. The reverse direction goes exactly as in the case $C = A$. ◀

▶ **Lemma 26.** Let $C \in \{A, I, S\}$. If π is a C-complete path in M_C starting in the initial state of M_C, then it is also a C-complete path in M starting in the initial state of M.

Proof. Let π be a C-complete path in M_C starting in its initial state. Since for all three variants, M_C has stricter conditions for actions being enabled than M does, π is guaranteed to exist in M. It remains to show that π is C-complete in M. This, however, is an immediate consequence of Proposition 25, from which it follows that a path starting in the initial state of M_C is $\mathcal{B}$-$\smile\!\!\bullet_C$-just exactly when that same path starting in the initial state of M is. ◀

We also wish to prove the other direction. However, it is not necessarily the case that any C-complete path in M is also a C-complete path in M_C, if only because not every path in M exists in M_C. Thus, we need a way to convert a path in M into a path that is guaranteed to exist in M_C. We define a rewrite relation on paths, where $\pi \rightsquigarrow \pi'$ denotes that π is rewritten into π'. Namely $\pi \rightsquigarrow \pi'$ if, and only if, there exists paths σ and ρ, states s and s', thread $t \in \mathbb{T}$, and an action b with $thr(b) \neq t$, such that

- either $\pi = \sigma\ sr_{t,r}\ s\ b\ \rho$ and $\pi' = \sigma\ b\ s'\ sr_{t,r}\ \rho$,
- or $\pi = \sigma\ sw_{t,r}(d)\ s\ b\ \rho$ and $\pi' = \sigma\ b\ s'\ sw_{t,r}(d)\ \rho$,
- or $\pi = \sigma\ b\ s\ fr_{t,r}(d)\ \rho$ and $\pi' = \sigma\ fr_{t,r}(d)\ s'\ b\ \rho$,
- or $\pi = \sigma\ b\ s\ fw_{t,r}\ \rho$ and $\pi' = \sigma\ fw_{t,r}\ s'\ b\ \rho$.

Thus, this rewrite relation moves any start read or start write action forwards by swapping it with an action b of another thread, and in the same manner moves any finish read or finish write action backwards. To establish that whenever a path π of any of the above forms exists in M, the corresponding π' also exists in M, it suffices to prove that for any transition $s \xrightarrow{b} s'$ that is part of a path π in M: (i) if a start action a is enabled in s and $thr(a) \neq thr(b)$, then a is enabled in s', and (ii) if a finish action a is enabled in s' and $thr(a) \neq thr(b)$, then a is enabled in s. The former follows directly from Lemma 6; (ii) straightforwardly holds in the case of a finish write action, and holds in the case of a finish read action because the return value is determined by the matching order read action, which belongs to the same thread. Since paths may be infinite, this rewrite system need not terminate, but in the limit it converts any path π into a normal form $norm(\pi)$ in which each action $or_{t,r}$ is immediately preceded by the corresponding $sr_{t,r}$ and immediately followed by the corresponding $fr_{t,r}(d)$,

and likewise for write actions. To convergence to this limit, one should give priority to rewrite steps that apply closer to the beginning of the path. This normal form also exists in M_A, M_I and M_S.

▶ **Lemma 27.** Let $C \in \{A, I, S\}$. If π is a C-complete path in M starting in the initial state of M, then $norm(\pi)$ is a C-complete path in M_C starting in the initial state of M_C.

Proof. Let π be a C-complete path in M starting in its initial state. Trivially, $norm(\pi)$ exists in M_C. It remains to show that $norm(\pi)$ is C-complete in M_C. Note that M_C is a subgraph of M. Hence, if $norm(\pi)$ is C-complete in M, then it is also C-complete in M_C. It therefore suffices to prove that $norm(\pi)$ is C-complete in M. This is an immediate consequence of Propositions 19–21, given that π and $norm(\pi)$ have the same sets of thread-enabled actions and $occ_\pi = occ_{norm(\pi)}$ (i.e., each action occurs as often in $norm(\pi)$ as it occurs in π). ◀

▶ **Theorem 28.** $M_A =^A_{WCT} M$, $M_I =^I_{WCT} M$ and $M_S =^S_{WCT} M$.

Proof. We prove that $WCT_C(M) = WCT_C(M_C)$ for all $C \in \{A, I, S\}$. We prove this by mutual set inclusion.

First, let π be a C-complete path from the initial state of M_C, such that $\ell^-(\pi) \in WCT_C(M_C)$. By Lemma 26, π is also a C-complete path from the initial state of M. Thus, $\ell^-(\pi) \in WCT_C(M)$.

Second, let π be a C-complete path from the initial state of M, such that $\ell^-(\pi) \in WCT_C(M)$. Then by Lemma 27, $norm(\pi)$ is a C-complete path in M_C. Note that $\ell^-(norm(\pi)) = \ell^-(\pi)$, hence $\ell^-(\pi) \in WCT_C(M_C)$.

We have proven $M_C =^C_{WCT} M$ for all $C \in \{A, I, S\}$. ◀

G mCRL2 models

In this section, we comment on some of the details of our thread and register models in mCRL2. For information on the mCRL2 language, we refer to [31]. We made several alterations while translating the process-algebraic definitions of MWMR safe, regular and atomic registers (see Appendix B) to mCRL2 processes. Some of these were necessary to obtain valid mCRL2 models, others were employed to reduce the state space of the models. The models are available as supplementary material. Note that the exact models differ slightly from those used in [51]; we altered them to align more closely with the process-algebraic definitions.

The most important differences between the definitions from Appendix B and the mCRL2 models are as follows:

- Since the three models do not require exactly the same information from the status object, and tracking unused information unnecessarily increases the state space of the model, we separate the status object into three variants, one for each register type, that only track the required information for that type.
- We reset values to a pre-defined default whenever we know that the value is no longer relevant. For example, the safe register model only needs to know the value that a thread intends to write if this write operation does not encounter an overlapping write; if there are multiple concurrent writes, it will not matter what their intended values were. Hence, in the safe register model we only track a single value instead of a mapping from threads to values for *rec*, and reset this value to its default whenever we observe zero or more than one active writer.

- For regular registers, instead of computing on the spot what the values of all active writes are whenever a read starts, we use a multiset to keep track of those values as writes start and finish; this is more straightforward and, since it adds no new information, does not expand the state-space.
- To further reduce the state space, we add the value of a new writer only to the *posv* of active readers, rather than all threads. Since *posv* is reset for a thread whenever it starts a read, this does not affect the behaviour of the model.
- We have moved summations inwards whenever possible, so that $\mathbb{D}_r$ is only summed over when the resulting value is actually relevant.

In addition to the registers, the threads must also be modelled in mCRL2. We already mentioned the relevant choices made here in Section 6. It remains to discuss how the parallel composition as defined in Section 2 and employed in Section 4 is modelled in mCRL2. In order to get the right communication between threads and registers, we create "sending" and "receiving" versions of all register interface actions, and define a communication function so that the two versions together form the correct action. For instance, to start a read of register r, thread t does the action $start\_read\_s(t, r)$. The register simultaneously does the action $start\_read\_r(t, r)$, and this communication appears in the model as $start\_read(t, r)$. The register and thread local actions do not need such a modification, since they are only performed by a single component.

It is worth noting that the mCRL2 version of the modal μ-calculus does not support quantification over actions directly, but does allow quantification over the data parameters of actions. In order to express the formulae we need (see Appendix H), we therefore add the action *label* to every action in the model, creating multi-actions. We give the *label* action a parameter over a set of labels $\mathbb{L}$, where each label corresponds to one of the action names used in Appendix B. We then hide the original actions, so that only *label* is visible in the model. This way, we can refer to the labels in our formulae, which circumvents the issue of not being able to quantify over actions. This approach is based on [14].

H Modal μ-calculus formulae

In [52], we presented template formulae that can be instantiated to capture liveness properties that fit Dwyer, Avrunin and Corbett's Property Specification Patterns (PSP) [22], incorporating a variety of completeness criteria, including justness.

In this appendix, we recap the three correctness properties we verified, and give the modal μ-calculus formulae for these properties. To be able to use the template formulae with justness for the liveness properties, we must show how they fit into PSP. We do not thoroughly explain the modal μ-calculus here; instead, we refer to [52] for more information on how these formulae should be understood. We do not use the mCRL2 modal μ-calculus syntax here, since it results in longer and less readable formulae. However, the files are all given in the supplementary material.

Mutual exclusion The mutual exclusion property says that at any given time, at most one thread will be in its critical section. In our models, a thread t accessing its critical section is represented by the action c_t. We reformulate the mutual exclusion property for our models as: for all threads t and t' such that $t \neq t'$, at any time it is impossible that both c_t and $c_{t'}$ are enabled. This is captured by the following modal μ-calculus formula:

$$\bigwedge_{t,t' \in \mathbb{T}} ((t \neq t') \Rightarrow [Act^\star]\neg(\langle c_t \rangle tt \wedge \langle c_{t'} \rangle tt)) \tag{3}$$

Deadlock freedom The deadlock freedom property says that whenever at least one thread is running its entry protocol, eventually some thread will enter its critical section. Since we want to apply the results from [52], we need to first show how this can be represented in PSP, using the actions in our model. This property fits into the global response pattern of [22]: whenever the trigger occurs, in this case one thread being in its entry protocol, we want the response to occur eventually, in this case some thread executing its critical section. We cannot directly apply the results from [52], however, since there we require the trigger to be a set of actions and a thread "being in the entry protocol" cannot simply be captured by the occurrence of an action. Instead, a thread t is in its entry protocol between the execution of nc_t and the subsequent execution of c_t. Fortunately, this requires only a minor deviation from the template given in [52], namely by allowing the trigger to be a regular expression over sets of actions instead.

Assume that for $C \in \{T, S, I, A\}$ and all actions a in our model, we have a function $\#_C\{a\}$ that maps an action to all actions that interfere with it according to concurrency relation C. We have encoded these functions in our mCRL2 models. For clarity, let $c_{all} = \bigcup_{t\in\mathbb{T}}\{c_t\}$.

In the modal μ-calculus formula, we capture that there *does not* exist a path from the initial state of the model that violates deadlock freedom and is just. This is equivalent to checking that deadlock freedom is satisfied on all just paths. The formula

$$\nu X.(\bigwedge_{a\in\overline{\mathcal{B}}}(\langle a\rangle tt \Rightarrow \langle\overline{c_{all}}^{\star}\rangle(\langle\#_C\{a\}\setminus c_{all}\rangle X)))$$

holds in a state s iff there is a path starting in s that (i) is just, in the sense that any state enabling an action $a \notin \mathcal{B}$ is followed by an action from $\#_C\{a\}$, and (ii) does not contain any action from c_{all}. In other words, this formula says that from a state s, there is a just path where the response never occurs. We then merely need to prepend the formula for an occurrence of the trigger, namely an occurrence of nc_t, for some $t \in \mathbb{T}$, that is not (yet) followed by c_t, and negate the resulting formula to capture that the initial state of the model does not admit just paths that violate deadlock freedom.

The final formula for deadlock freedom under $JA_{\mathcal{B}}^{\smile\bullet C}$ is:

$$\neg\langle Act^{\star}\rangle\bigvee_{t\in\mathbb{T}}(\langle nc_t\cdot\overline{c_t}^{\star}\rangle\nu X.(\bigwedge_{a\in\overline{\mathcal{B}}}(\langle a\rangle tt \Rightarrow \langle\overline{c_{all}}^{\star}\rangle(\langle\#_C\{a\}\setminus c_{all}\rangle X)))) \tag{4}$$

Starvation freedom Starvation freedom says that whenever a thread leaves its non-critical section, it will eventually enter its critical section. Unlike deadlock freedom, starvation freedom does fit into the global response pattern using only sets of actions: the trigger is the occurrence of nc_t for some $t \in \mathbb{T}$, and the response is the occurrence of c_t. Using the same functions $\#_C\{a\}$ for $C \in \{T, S, I, A\}$ and actions a as the deadlock freedom formula, we get the following modal μ-calculus formula for starvation freedom under $JA_{\mathcal{B}}^{\smile\bullet C}$:

$$\bigwedge_{t\in\mathbb{T}}(\neg\langle Act^{\star}\cdot nc_t\rangle\nu X.(\bigwedge_{a\in\overline{\mathcal{B}}}(\langle a\rangle tt \Rightarrow \langle\overline{c_t}^{\star}\rangle(\langle\#_C\{a\}\setminus c_t\rangle X)))) \tag{5}$$

I Algorithms

In this appendix, we go over all the algorithms mentioned in Table 1 that have not already been discussed in Section 6. We give the pseudocode of algorithms, and where relevant also discuss the results of the verification.

I.1 Anderson's algorithm

Anderson's algorithm is presented in [4]. It only works for 2 threads, and has different code for both threads. For clarity, we break from our established shorthand of $j = 1 - i$ for just this algorithm, and present the algorithms for $i = 0$ (Algorithm 5) and $j = 1$ (Algorithm 6) separately. There are six Booleans total: $p[i], p[j], q[i], q[j], t[i]$ and $t[j]$. All are initialised to *true*.

Algorithm 5 Anderson's algorithm for $i = 0$

```
1:  p[i] ← false
2:  q[i] ← false
3:  x ← t[j]
4:  t[i] ← x
5:  if x = true then
6:      p[i] ← true
7:      await p[j] = true
8:  else
9:      q[i] ← true
10:     await q[j] = true
11: critical section
12: p[i] ← true
13: q[i] ← true
```

Algorithm 6 Anderson's algorithm for $j = 1$

```
1:  p[j] ← false
2:  q[j] ← false
3:  x ← ¬t[i]
4:  t[j] ← x
5:  if x = true then
6:      q[j] ← true
7:      await p[i] = true
8:  else
9:      p[j] ← true
10:     await q[i] = true
11: critical section
12: p[j] ← true
13: q[j] ← true
```

In accordance with Anderson's claim, the algorithm satisfies all three properties with non-atomic as well as atomic registers.

I.2 Attiya-Welch's algorithm

What we call the Attiya-Welch algorithm is presented in both [9] (original presentation, Algorithm 7) and [50] (variant presentation, Algorithm 8) as a variant of Peterson's algorithm. The two presentations of the algorithm have different behaviour, so we present both. This algorithm is only defined for $N = 2$. Each thread i has a Boolean $flag[i]$, initialised to *false*. There is also a global variable *turn* over $\mathbb{T}$, initialised to 0.

Algorithm 7 Attiya-Welch algorithm, orig.

```
1:  flag[i] ← false
2:  await flag[j] = false ∨ turn = j
3:  flag[i] ← true
4:  if turn = i then
5:      if flag[j] = true then
6:          goto line 1
7:  else
8:      await flag[j] = false
9:  critical section
10: turn ← i
11: flag[i] ← false
```

Algorithm 8 Attiya-Welch algorithm, var.

```
1:  repeat
2:      flag[i] ← false
3:      await flag[j] = false ∨ turn = j
4:      flag[i] ← true
5:  until turn = j ∨ flag[j] = false
6:  if turn = j then
7:      await flag[j] = false
8:  critical section
9:  turn ← i
10: flag[i] ← false
```

In the original presentation of the Attiya-Welch algorithm, no claims are made about

its correctness with non-atomic registers. It is therefore perhaps surprising to note that it does satisfy many properties with non-atomic registers. In [51], we showed that the original presentation of the Attiya-Welch algorithm satisfies reachability of the critical section, a property weaker than deadlock freedom, with both safe and regular registers. Here, we show that while it satisfies starvation freedom with regular registers, it only satisfies deadlock freedom with safe registers. An execution violating starvation freedom with safe registers is the following:

- Thread 0 runs through lines 1 through 9 without competition; since $\mathit{flag}[1] = \mathit{false}$ it can reach line 9 without problem. On line 10, it starts writing 0 to *turn*, which is already 0.
- Thread 1 executes line 1, setting $\mathit{flag}[1]$ to *false*, which it already was. On line 2, it reads $\mathit{flag}[0] = \mathit{true}$ and $\mathit{turn} = 1$. Note that the value 1 has never been written to *turn*, but due to the read overlapping with thread 0's write, the value can still be read. Thread 1 can therefore not proceed through line 2.
- Thread 0 finishes the exit protocol, setting *turn* to 0 and $\mathit{flag}[0]$ to *false*.

At this point, we have $\mathit{flag}[0] = \mathit{flag}[1] = \mathit{false}$ and $\mathit{turn} = 0$, the same values the variables had at the start. Hence, thread 0 can reach line 10 again, without interference by thread 1. By having thread 1 always read $\mathit{flag}[0]$ and *turn* at exactly the wrong time, it will remain forever in line 2, without ever reaching the critical section.

Note that this execution relies on reading a value that has never been written. We can adjust the algorithm slightly, so that *turn* is read before it is written, and only updated if it does not already have the intended value. Starvation freedom is then satisfied with safe registers.

We find that the Attiya-Welch algorithm does not satisfy starvation freedom with atomic registers under $JA_{\mathcal{B}}^{\bullet\frown s}$. This violation is rather trivial: as long as $\mathit{flag}[1] = \mathit{false}$, thread 0 can infinitely often execute the algorithm to get to its critical section. During this execution, it will write to $\mathit{flag}[0]$ repeatedly, preventing thread 1 from reading $\mathit{flag}[0]$ on line 2. Hence, thread 1 can be prevented from ever reaching the critical section if a write can block a read. Note that on line 2 of the algorithm, a thread already wants access to its critical section but has in no way communicated this to the other thread. This is not something we can fix by merely slightly altering a condition or reading a variable before writing, it would require more significant alterations to the algorithm so that a thread has to communicate its intention before attempting to read a register another thread can infinitely often write to. We do not explore such alterations here.

Instead, we turn our attention to the variant presentation from [50]. In this version, the goto statements have been eliminated.

In [50], it is claimed that the algorithm satisfies all three properties for all four interpretations of MWMR regular registers proposed in that paper. A comparison between their definitions and our regular register model is given in [51]; here it suffices to observe that their weakest definition is weaker than our regular register model. Hence, we would expect all three properties to be satisfied by our regular model. As can be observed in Table 1, this is not the case. While the two presentations of the algorithm are seemingly equivalent, the altered pseudocode suggests the *turn* variable needs to be read twice in a row, where it is read only once on line 4 of the original presentation. This allows new-old inversion to occur with the non-atomic register models, and causes the variant presentation to no longer satisfy deadlock freedom with non-atomic registers. An execution demonstrating the violation is given in [51]. If we alter the model so that the value of *turn* is only read once for the two conditions, we see the expected behaviour, given the results in [50]. If we then also make the adjustment that *turn* is only written to when its value would actually change, we see the

same behaviour as our altered version of the original presentation. The issue of starvation freedom not being satisfied for atomic registers with $JA_{\mathcal{B}}^{\smile \bullet s}$ is also present in this variant.

I.3 Burns-Lynch's algorithm and Lamport's 1-bit algorithm

The Burns-Lynch algorithm ([18], Algorithm 9) and Lamport's 1-bit algorithm ([39], Algorithm 10) are two very similar algorithms. In [39], Lamport makes the explicit claim that this algorithm satisfies mutual exclusion and deadlock freedom with safe registers. Both algorithms are designed for an arbitrary N, and both use only a single shared Boolean per thread, initialised to *false*. For the sake of easy comparison, we name this Boolean $flag[i]$ for each thread i for both algorithms, although Lamport uses the name $x[i]$ and Burns and Lynch use $Flag[i]$.

Algorithm 9 The Burns-Lynch algorithm

```
1: repeat
2:     flag[i] ← false
3:     await ∀_{j<i} : flag[j] = false
4:     flag[i] ← true
5: until ∀_{j<i} : flag[j] = false
6: await ∀_{j>i} : flag[j] = false
7: critical section
8: flag[i] ← false
```

Algorithm 10 Lamport's 1-bit algorithm

```
1: flag[i] ← true
2: for j from 0 to i − 1 do
3:     if flag[j] = true then
4:         flag[i] ← false
5:         await flag[j] = false
6:         goto line 1
7: for j from i + 1 to N − 1 do
8:     await flag[j] = false
9: critical section
10: flag[i] ← false
```

Neither algorithm was designed to satisfy starvation freedom, only deadlock freedom. In [15], it is claimed that both work with non-atomic registers, albeit without satisfying starvation freedom. With the Attiya-Welch algorithm, we saw that minor differences in presentation can impact the correctness of an algorithm. This does not appear to be the case with these two algorithms; they show the same, and the expected, behaviour.

I.4 From deadlock freedom to starvation freedom

We here give the pseudocode for the algorithm for turning a deadlock-free algorithm into a starvation-free one, as discussed in Section 6.5. See Algorithm 11 for the algorithm as presented in [28]. It works for arbitrary N.

In addition to the registers used by the deadlock-free algorithm, this algorithm uses a Boolean *flag* for every thread, initialised to *false* and a shared register over $\mathbb{T}$ called *turn*, which here is initialised to 0. Naturally, these registers must be distinct from those used in the deadlock-free algorithm.

I.5 Dijkstra's algorithm

Dijkstra's algorithm [21] is given as Algorithm 12. It works for arbitrary N. Every thread i has two Booleans: $b[i]$ and $c[i]$, both initialised to *true*. There is also a global register k over $\mathbb{T}$, initialised at 0.

We find that this algorithm never satisfies starvation freedom, which is unsurprising since Dijkstra did not require that property in his original presentation of the mutual exclusion problem.

Algorithm 11 Algorithm for making a deadlock-free solution starvation-free

```
1: flag[i] ← true
2: repeat
3:     tmp ← turn
4: until tmp = i ∨ flag[tmp] = false
5: entry protocol of deadlock-free algorithm
6: critical section
7: flag[i] ← false
8: tmp ← turn
9: if flag[tmp] = false then
10:    turn ← (tmp + 1) mod N
11: exit protocol of deadlock-free algorithm
```

Algorithm 12 Dijkstra's algorithm

```
1: b[i] ← false
2: if k ≠ i then
3:     c[i] ← true
4:     if b[k] = true then
5:         k ← i
6:     goto line 2
7: else
8:     c[i] ← false
9:     for j from 0 to N − 1 do
10:        if j ≠ i ∧ c[j] = false then
11:            goto line 2
12: critical section
13: c[i] ← true
14: b[i] ← true
```

I.6 Kessels's algorithm

Kessels's algorithm (Algorithm 13) is a variant on Peterson's, presented in [35]. Its basic form is designed for $N = 2$, and each thread i has two variables: a Boolean $q[i]$, initialised at *false*, and $r[i]$ over $\mathbb{T}$, initialised at 0.

Algorithm 13 Kessels's algorithm

```
1: q[j] ← true
2: r[j] ← (r[i] + j) mod 2
3: await q[i] = false ∨ (r[j] ≠ ((r[i] + j) mod 2))
4: critical section
5: q[j] ← false
```

Since it is a variant on Peterson's, merely with only single-writer registers, it is unsurprising that the two algorithms give the same results to our verification. The violating execution for regular registers, like many of the executions discussed in this section, relies on new-old inversion.

- Thread 0 sets $q[1]$ to *true*, and then reads $r[0] = 0$. It therefore starts writing 1 to $r[1]$,

which was 0.

- Thread 1 sets $q[0]$ to *true*, and then reads $r[1]$ with overlap, reading the new value 1. It therefore writes 1 to $r[0]$. It then reads $r[0] = 1$ and $r[1] = 0$, the latter being the old value. Since $1 \neq (0 + 0) \mod 2$, it can reach its critical section.
- Thread 0 finishes its write to $r[1]$, and on line 3 reads $r[1] = 1$ and $r[0] = 1$; since $1 \neq (1 + 1) \mod 2$, it can reach its critical section.

This violation does not contradict any claims made by Kessels in [35], who said nothing about the correctness of this algorithm with non-atomic registers. Rather, we find the violation interesting because it clearly illustrates that using only single-writer Booleans does not automatically mean that an algorithm is robust to non-atomic registers.

I.7 Knuth's algorithm

Knuth's algorithm (Algorithm 14) comes from [36]. It works for arbitrary N. Every thread i has a register $control[i]$, which is over $\{0, 1, 2\}$, initialised to 0. Additionally, there is a global register k over $\mathbb{T}$, also initialised to 0.[9]

Algorithm 14 Knuth's algorithm

```
 1: control[i] ← 1
 2: for j from k downto 0 do
 3:     if j = i then
 4:         goto line 12
 5:     if control[j] ≠ 0 then
 6:         goto line 2
 7: for j from N−1 downto 0 do
 8:     if j = i then
 9:         goto line 12
10:     if control[j] ≠ 0 then
11:         goto line 2
12: control[i] ← 2
13: for j from N−1 downto 0 do
14:     if j ≠ i ∧ control[j] = 2 then
15:         goto line 1
16: k ← i
17: critical section
18: if i = 0 then
19:     k ← N−1
20: else
21:     k ← i−1
22: control[i] ← 0
```

The goal of Knuth's algorithm is to improve upon Dijkstra's result by having starvation freedom in addition to deadlock freedom. This goal is indeed accomplished with atomic

[9] In Knuth's original presentation [36], k was initialised to -1, which there was called 0, as the threads ranged from 1 to N. This difference is immaterial; when thread 1 goes once through the algorithm without contention, k is set to 0, so this could just as well be the initial value.

registers, as we see in Table 1. Knuth makes no claims on this algorithm's behaviour with non-atomic registers. However, it is interesting that there is a difference between the behaviour with safe and regular registers.

An execution violating deadlock freedom for two threads and safe registers is as follows:

- Thread 1 starts setting *control*[1] to 1
- Thread 0 starts the competition; since $k = 0$, it goes to line 12 and sets *control*[0] to 2. When it reads *control*[1] on line 14, it reads a 2, which has never been written. Hence, it has to return to line 1, and hence start writing to *control*[0].
- Thread 1 finishes line 1 and goes to line 2, where it has to read *control*[0]. It reads *control*[0] = 0, not the new or old value, and hence goes to line 7. There, since $N - 1 = 1$, it goes to line 12 and sets *control*[1] to 2. It then goes to line 13, where it finds that *control*[0] = 2, due to the overlap with thread 0's write. Hence, it has to return to line 1, where it starts writing to *control*[1] again.

At this point, we are back where we started, with both threads at the beginning of the algorithm. We see there is a loop where both threads continually make progress, but always get sent back on line 15. Unlike with the Attiya-Welch algorithm, we cannot fix this by merely ensuring that a register is only written to when its value would change.

I.8 Lamport's 3-bit algorithm

We already discussed Lamport's 1-bit algorithm, but in [39] he also presented the 3-bit algorithm (Algorithm 15), which was designed to improve on the 1-bit algorithm by making it starvation-free. It works for arbitrary N. Every thread i has three Booleans: $x[i], y[i]$ and $z[i]$, all initialised to *false*.

This algorithm uses some auxiliary operations. The operation $\mathrm{ORD}(S)$ returns for a set $S \subseteq \mathbb{T}$ a list of all elements in S ordered from smallest to largest. Note that such an ordering is defined for $\mathbb{T}$ since, as we stated in Section 6, we use natural numbers to represent thread identifiers. This list is formalised as an increasing function $\gamma : \{1, \ldots, M\} \to S$, where $M = |S|$. We write $\mathrm{domain}(\gamma)$ for $\{1, \ldots, M\}$ and $\mathrm{range}(\gamma)$ for S. Additionally, there is the Boolean function $\mathrm{CG}(v, l)$, where $v : \{1, \ldots, M\} \to \mathbb{B}$ is a Boolean function mapping an index in $\{1, \ldots, M\}$ (denoting an element of S) to either *true* or *false*, and $l \in \{1, \ldots, M\}$.

$$\mathrm{CG}(v, l) \stackrel{\text{def}}{=} (v(l) = \mathrm{CGV}(v, l))$$

$$\mathrm{CGV}(v, l) \stackrel{\text{def}}{=} \begin{cases} \neg v(l{-}1) & \text{if } l > 1 \\ v(M) & \text{if } l = 1 \end{cases}$$

We write "**for** i **from** j **cyclically to** k" to mean an iteration that begins with $i = j$, then increments i by 1, taking the result modulo N. The iteration terminates when $i = k$, without executing the loop with $i = k$. We use $\oplus$ for addition modulo N.

The definitions of ORD, CG and CGV given above differ from those given by Lamport in [39]. He works with *cycles*, rather than lists, and defines CG and CGV based on indexed elements of those cycles, rather than using the indices directly as we do. We believe that our presentation is equivalent to the one given by Lamport, but find it more straightforward to explain. This presentation also aligns better with how the algorithm would be implemented, and indeed aligns more closely with our mCRL2 model.

In addition to these changes in presentation, we also made one change to the algorithm itself: line 4 was added by us to emphasise that the mapping ζ is a snapshot of the variables $z[i]$ for all $i \in \mathrm{range}(\gamma)$, and that the registers themselves do not get repeatedly read on line 5. For this, we introduce the operation SAVE_z^γ that creates a mapping from $\mathrm{domain}(\gamma)$

to the Booleans, such that for all $h \in \text{domain}(\gamma)$, $\text{SAVE}_z^\gamma(h) = z[\gamma(h)]$. Lamport does not state explicitly in [39] that the z registers must not be read repeatedly when computing the minimum, but if the registers are non-atomic and re-read during the computation it is possible that no minimum will be found. This observation is also made in [51].

Algorithm 15 Lamport's 3-bit algorithm

```
 1: y[i] ← true
 2: x[i] ← true
 3: γ ← ORD{j | y[j] = true}
 4: ζ ← SAVE_z^γ
 5: f ← γ(minimum{h ∈ domain(γ) | CG(ζ, h) = true})
 6: for j from f cyclically to i do
 7:     if y[j] = true then
 8:         if x[i] = true then x[i] ← false
 9:         goto line 3
10: if x[i] = false then goto line 2
11: for j from i ⊕ 1 cyclically to f do
12:     if x[j] = true then goto line 3
13: critical section
14: if z[i] = true then
15:     z[i] ← false
16: else
17:     z[i] ← true
18: x[i] ← false
19: y[i] ← false
```

I.9 Peterson's algorithm

Peterson's algorithm is already given in Section 6. We merely note here that an execution showing that Peterson's algorithm does not satisfy mutual exclusion with non-atomic registers is given in [51].

I.10 Szymanski's flag algorithm

We discussed Szymanski's 3-bit linear wait algorithm from [54] in Section 6.4. Earlier, in [53], Szymanski presented the flag algorithm, which we here give as Algorithm 16. Note that we give the pseudocode with a minor fix of an obvious typo: on line 10, we use a $\vee$ instead of a $\wedge$. In its original presentation, the flag algorithm uses a single integer variable *flag* per thread ranging over $\{0, 1, 2, 3, 4\}$, initially 0. Just like the 3-bit algorithm, it is designed for arbitrary N.

It is claimed that, while the integer version of the algorithm is not correct with non-atomic registers, by converting the integers to three Booleans each, *door_in*, *door_out* and *intent*, the algorithm can be made correct for non-atomic registers. According to [53], this translation should be done according to Table 2. This results in Algorithm 17. The three Booleans are all initialised as *false*.

We analysed both variants in [51], and found that neither algorithm is correct with non-atomic registers. Executions demonstrating these violations are provided there. Additionally, we observed in [51] that the Boolean variant of the algorithm violates mutual exclusion

even with atomic registers. We observed that the violating execution reported by mCRL2, given in full in [51], relies on the exit protocol resetting the three Booleans in the order $intent - door\_in - door\_out$. This is the order that is suggested by the final pseudocode provided by Szymanski in [53, Figure 3]. We posited in [51] that the violation of mutual exclusion could be fixed by changing the order to be $door\_out - intent - door\_in$ instead. Now that we can verify the liveness properties as well, we can confirm that this is indeed true: if this alternate exit protocol is used, the Boolean version of the flag protocol is equivalent to the integer version with respect to the properties verified in this paper.

Algorithm 16 Szymanski's flag algorithm

1: $flag[i] \leftarrow 1$
2: **await** $\forall_j : \ flag[j] < 3$
3: $flag[i] \leftarrow 3$
4: **if** $\exists_j : \ flag[j] = 1$ **then**
5: $\quad flag[i] \leftarrow 2$
6: $\quad$ **await** $\exists_j : \ flag[j] = 4$
7: $flag[i] \leftarrow 4$
8: **await** $\forall_{j<i} : \ flag[j] < 2$
9: **critical section**
10: **await** $\forall_{j>i} : \ flag[j] < 2 \vee flag[j] > 3$
11: $flag[i] \leftarrow 0$

Table 2 Translating the register *flag* to three Boolean registers *intent*, *door_in* and *door_out*.

flag	*intent*	*door_in*	*door_out*
0	*false*	*false*	*false*
1	*true*	*false*	*false*
2	*false*	*true*	*false*
3	*true*	*true*	*false*
4	*true*	*true*	*true*

Algorithm 17 Szymanski's flag algorithm implemented with Booleans

1: $intent[i] \leftarrow true$
2: **await** $\forall_j : \ intent[j] = false \vee door\_in[j] = false$
3: $door\_in[i] \leftarrow true$
4: **if** $\exists_j : \ intent[j] = true \wedge door\_in[j] = false$ **then**
5: $\quad intent[i] \leftarrow false$
6: $\quad$ **await** $\exists_j : \ door\_out[j] = true$
7: **if** $intent[i] = false$ **then** $intent[i] \leftarrow true$
8: $door\_out[i] \leftarrow true$
9: **await** $\forall_{j<i} : \ door\_in[j] = false$
10: **critical section**
11: **await** $\forall_{j>i} : \ door\_in[j] = false \vee door\_out[j] = true$
12: $intent[i] \leftarrow false$
13: $door\_in[i] \leftarrow false$
14: $door\_out[i] \leftarrow false$