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[1]
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.

Digital Object Identifier 10.4230/LIPIcs.CONCUR.2025.22

Related Version *Full Version*: `https://arxiv.org/abs/2507.13198` [28]

Supplementary Material *Model*: `The mCRL2 models and the modal μ-calculus formulae can be found in the mCRL2 GitHub repository (commit c6ad29e).`

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*,[2] 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,

[1] Corresponding author
[2] 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.

© Rob van Glabbeek, Bas Luttik and Myrthe S.C. Spronck;
licensed under Creative Commons License CC-BY 4.0

the thread is executing its *entry protocol*, the part of the mutual exclusion algorithm in 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* [38].

Memory models.[3] 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 [20]. 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* [18].

Deviating from Dijkstra's original presentation, several authors have considered a variation of the mutual exclusion problem where the atomicity assumption is dropped [36, 47, 37, 38, 52, 53, 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.

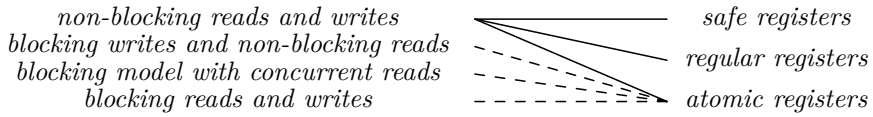
[3] A *memory model* describes the interactions of threads through memory and their shared use of the data. The models reviewed here differ in the degree in which different register accesses exclude each other, and in what values a register may return in case of overlapping reads and writes. In this 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.

In [39, 40], Lamport proposes a hierarchy of three memory models in this context, 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.

These three memory models 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*,[4] 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 [50], we checked the mutual exclusion property of several algorithms, with safe, regular and atomic MWMR registers, through model checking with the mCRL2 toolset [16]. 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

[4] In this terminology, from [18], 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[5] 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 [42, 5, 27]. In effect, making a fairness assumptions amounts to assuming that if something is tried often enough, it will always eventually succeed [27]. 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* [27, 24, 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 [38], "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.[6] 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 [36], 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 $N \in \mathbb{N}$,

[5] 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.

[6] A good illustration of unreliable behavioural reasoning is given in [25, 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 [52], 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 [19, 3] and its RW-safe variant [15], and Szymanski's 3-bit linear wait algorithm [53]. 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 $(\mathcal{S}, \mathit{Act}, \mathit{init}, \mathit{Trans})$ in which $\mathcal{S}$ is a finite set of states, Act is a finite set of actions, $\mathit{init} \in \mathcal{S}$ is the initial state, and $\mathit{Trans} \subseteq \mathcal{S} \times \mathit{Act} \times \mathcal{S}$ is a transition relation. We write $s \xrightarrow{a} s'$ for $(s, a, s') \in \mathit{Trans}$. We say an action a is *enabled* in a state s if there exists a state s' such that $(s, a, s') \in \mathit{Trans}$.

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 $a_1, a_2, \ldots \in \mathit{Act}$, 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 [33], 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 $P_i = (\mathcal{S}_i, \mathit{Act}_i, \mathit{init}_i, \mathit{Trans}_i)$ 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 $P = (\mathcal{S}, \mathit{Act}, \mathit{init}, \mathit{Trans})$ in which $\mathcal{S} = \mathcal{S}_1 \times \ldots \times \mathcal{S}_k$, $\mathit{Act} = \bigcup_{1 \leq i \leq k} \mathit{Act}_i$, $\mathit{init} = (\mathit{init}_1, \ldots, \mathit{init}_k)$, and a transition $((s_1, \ldots, s_k), a, (s'_1, \ldots, s'_k))$ is in Trans if, and only if, $a \in \mathit{Act}$ and the following are true for all $1 \leq i \leq k$: if $a \notin \mathit{Act}_i$, then $s_i = s'_i$, and if $a \in \mathit{Act}_i$, then $(s_i, a, s'_i) \in \mathit{Trans}_i$.
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* [24, 27]. 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 $(\mathcal{S}, \mathit{Act}, \mathit{init}, \mathit{Trans})$, a relation $\smile\!\!\!\bullet \subseteq \mathit{Act} \times \mathit{Act}$ is a *concurrency relation* if, and only if:

- $\smile\!\!\!\bullet$ is irreflexive.
- For all $a \in \mathit{Act}$, 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 $b \in \mathit{Act}$ 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 [50], 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 A; 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 [48]. 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 [50] we present one extension and compare it to four different suggestions from [49]. The complexity comes from determining what the last written value is, given that writes may be concurrent with each other. Here, following [50], 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 B.

▶ **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 [28, 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 [21, 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 [28, Appendix E], we formally characterise just paths in our thread-register models for all six variants. In [28, 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, 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 [30]. 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 [28, 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 [16]. 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 [51] to incorporate the justness assumption into our formulae for deadlock freedom and starvation freedom. The full modal μ-calculus formulae appear in [28, Appendix H] (and also as supplementary material). Besides the correctness properties discussed in Section 1, Dijkstra [20] 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.". For the algorithms we discuss in detail, we include their pseudocode. 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. Pseudocode and further discussion of the algorithms not covered here appear in [28, 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 [23, 25]. 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 [25].

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.) [49]	2	M	M	S	D	M	M
Attiya-Welch (var., alt.)	2	S	S	S	D	M	M
Burns-Lynch [17]	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 [15]	2	S	S	S	D	M	M
Dekker RW-safe (DFtoSF)	2	S	S	S	S	M	M
Dijkstra [20]	3	M	D	D	M	M	M
Kessels [34]	2	X	X	S	S	M	M
Knuth [35]	3	M	S	S	M	M	M
Lamport 1-bit [38]	3	D	D	D	D	M	M
Lamport 1-bit (DFtoSF)	3	S	S	S	S	M	M
Lamport 3-bit [38]	3	S	S	S	S	M	M
Peterson [46]	2	X	X	S	S	M	M
Szymanski flag (int.) [52]	3	X	X	S	S	M	M
Szymanski flag (bit) [52]	3	X	X	X	X	X	X
Szymanski 3-bit lin. wait [53]	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 [46], 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 [19]. 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 [15]. 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 [40]. 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 [15] 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 [45], 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 [15] 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 [15] 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 [15], 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 [53], 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 [50].

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

registers, given in detail in [50], 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 [44]. 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 [48, 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 [54]. We take the pseudocode from [26], 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 [38], which (by design) does not satisfy starvation freedom at all, and to the RW-safe version of Dekker's algorithm [15], 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.

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, 43, 29]. Non-atomic registers have been covered less frequently, and their verification is often restricted to single-writer registers [41, 15]. The safety properties of mutual exclusion algorithms with MWMR non-atomic registers were verified using mCRL2 in [50]. In several papers by Nigro, including [44, 45], 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, 22, 55, 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, 31, 32] 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. Dekker's mutual exclusion algorithm made RW-safe. *Concurrency and Computation: Practice and Experience*, 28(1):144–165, 2016. doi:10.1002/cpe.3659.

16 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.

17 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.

18 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.

19 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.

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

21 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.

22 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.

23 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.

24 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.

25 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.

26 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.

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

28 Rob J. van Glabbeek, Bas Luttik, and Myrthe S.C. Spronck. Just verification of mutual exclusion algorithms, 2025. Full version of the present paper, available at https://arxiv.org/abs/2507.13198.

29 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.

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

31 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.

32 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.

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

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

35 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.

36 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.

37 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.

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

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

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

41 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.

42 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.

43 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.

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

45 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.

46 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.

47 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.

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

49 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.

50 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.

51 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.

52 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.

53 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.

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

55 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 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.

A.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:

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

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:

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

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}:

$$stor(init) = d_{init} \quad wrts(init) = \emptyset \quad rec(init, t) = d_{init} \quad posv(init, t) = \emptyset$$
$$rds(init) = \emptyset \quad pend(init) = \emptyset \quad ovrl(init, t) = false$$

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:

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

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

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

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

$$wrts(s') = wrts(s) \cup \{t\} \qquad rec(s', t) = d \qquad ovrl(s', t) = (wrts(s) > 0)$$
$$pend(s') = pend(s) \cup \{t\} \quad posv(s', t') = posv(s, t') \cup \{d\} \quad ovrl(s', t') = true$$

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

$$stor(s') = d \qquad wrts(s') = wrts(s) \setminus \{t\}$$

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

$$pend(s') = pend(s) \setminus \{t\} \qquad rec(s', t) = stor(s)$$

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

$$stor(s') = d \qquad pend(s') = pend(s) \setminus \{t\}$$

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.

A.2 Safe MWMR registers

See Figure 1 for the process equation representing our safe register model.

The correspondence between the process and the four rules given for MWMR safe registers in Subsection 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 1 Safe register process

A.3 Regular MWMR registers

See Figure 2 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.

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.

A.4 Atomic MWMR registers

See Figure 3 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.

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

Figure 2 Regular register process

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

Figure 3 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.

B Thread consistency proof

In the [28, Appendix C], we provide a detailed proof of Lemma 6. Here, we give a less detailed version.

▶ **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. Consider a thread-register model $M = (\mathcal{S}, Act, init, Trans, thr, reg)$ and a specific state $s \in \mathcal{S}$. Let a be an arbitrary action enabled in s and assume there exists a transition $s \xrightarrow{b} s'$ to some $s' \in \mathcal{S}$ such that $thr(a) \neq thr(b)$. We must prove that a is enabled in s'. Let $thr(a) = t$ and $reg(a) = r$. Note that a is either a thread local action, or a register (local or interface) action. We consider both cases.

- If a is a thread local action, then it is enabled whenever a is enabled in the LTS belonging to t. Since b is an action by a different thread, the local state of the LTS of t is not affected by b, and so is the same in both s and s'. Thus, a is still enabled in s'.
- If a is not a thread local action, then $r \neq \bot$ and a involves the register r. Note that if $reg(b) \neq r$, then b can affect neither the local state of the LTS of t nor the local state of the LTS of r, so a is guaranteed to still be enabled in s'. We proceed under the assumption that $reg(b) = r$. It is still the case that b cannot affect the local state of the LTS associated with t, so this cannot prevent a from being enabled in s'. It therefore remains to prove that a is enabled in the local state of the LTS associated with r in s'. Let s_r be the local state of the LTS associated with r in s, and s'_r be its equivalent in s'. We proceed by considering which action a is.
  - If a is $sr_{t,r}$, $sw_{t,r}(d)$ for some $d \in \mathbb{D}_r$, $fw_{t,r}$, $or_{t,r}$ or $ow_{t,r}$, then whether it is enabled in state s' is only dependent on t's presence in a subset of $rds(s'_r)$, $wrts(s'_r)$ and $pend(s'_r)$. Adding t to or removing t from these sets can only be done by update functions that belong to actions c with $thr(c) = t$. Since $thr(b) \neq t$, the inclusion of t in these sets is unaltered between s_r and s'_r. Hence, since a is enabled in s_r, it is also enabled in s'_r.

- It remains to consider the case that a is $fr_{t,r}(d)$ for some $d \in \mathbb{D}_r$. We must show that, even if b is an operation on the register r, t can still read the value from r in s' that it could read in s. We do a case distinction on the type of register r.
  - If r is a safe register, then d can be read in s because $ovrl(s_r, t)$ or $stor(s_r) = d$. In the former case, regardless of which action b is, it will still be the case that $ovrl(s'_r, t)$ holds, so t can still return d. In the latter case, while it is possible that b changes the stored value of r if $b = fw_{thr(b),r}$, this means that $thr(b)$ is an active writer while t is an active reader, so there is overlap. Thus, $stor(s'_r) = d$ or $ovrl(s'_r, t)$ is true. Either way, t can still return d in s'.
  - If r is a regular register, then d is in $posv(s_r, t)$. The set of possible values only grows as long as a read is active; it only resets when a new read starts. Since b cannot be a start read by t, d is in $posv(s'_r, t)$, and so t can return d in s'.
  - If r is atomic, then when the finish read action a occurs, it simply returns the value that was previously recorded when the read was ordered. Since $thr(b) \neq t$, $rec(s_r, t) = rec(s'_r, t)$. Therefore t can still return d in s'.

In each case, we have shown that a is enabled in s'. ◀