Process-Algebraic Models of Multi-Writer Multi-Reader Non-Atomic Registers

Myrthe Spronck
Eindhoven University of Technology, The Netherlands

Bas Luttik
Eindhoven University of Technology, The Netherlands

Abstract

We present process-algebraic models of multi-writer multi-reader safe, regular and atomic registers. We establish the relationship between our models and alternative versions presented in the literature. We use our models to formally analyse by model checking to what extent several well-known mutual exclusion algorithms are robust for relaxed atomicity requirements. Our analyses refute correctness claims made about some of these algorithms in the literature.

2012 ACM Subject Classification Theory of computation → Shared memory algorithms; Theory of computation → Verification by model checking

Keywords and phrases mutual exclusion, model checking, non-atomic reads and writes, regular register.

Acknowledgements We thank Rob van Glabbeek for insightful discussions on the topic of this paper.

1 Introduction

The mutual exclusion problem was first outlined by Dijkstra [9]. Given n threads executing some code with a special section called the "critical section", the problem is to ensure that at any one time at most one of the threads is executing its critical section. Dijkstra explicitly states that communication between threads should be done through shared registers, and that reading from and writing to these registers should be considered atomic operations; when two threads simultaneously interact with the register, be it through reading or writing, the register behaves as though these operations took place in some total order.

Lamport argued that solutions to the mutual exclusion problem that assume atomicity of register operations do not fundamentally solve it [19]. After all, implementing atomic operations would require some form of mutual exclusion at a lower level. Many algorithms have been proposed that solve the mutual exclusion problem without requiring atomicity of register operations, most famously Lamport's own Bakery algorithm [17].

Analysing distributed algorithms using non-atomic registers for communication between threads can be difficult, and correctness proofs are error-prone. Due to the vast number of execution paths of distributed algorithms, especially when overlapping register operations need to be taken into account, manual correctness proofs are likely to miss issues. One better uses computer tools (e.g., model checkers or theorem provers) to support correctness claims with a detailed and preferably exhaustive analysis. This introduces the need for formal models of non-atomic registers.

Lamport proposed a general mathematical formalism for reasoning about the behaviour of concurrent systems that do not rely on the atomicity of operations, which he then uses to analyse the correctness of four solutions to the mutual exclusion problem not relying on atomicity [19, 18]. In [20], he studies in more detail the notion of single-writer multi-reader (SWMR) non-atomic register to implement communication between concurrent threads of computation; there, he distinguishes two variants, which he refers to as *safe* and *regular*. When a read operation to a SWMR safe register does not overlap with any write operations,

then it will return the value stored in the register, but when it does overlap with a write operation then it may return a completely arbitrary value in the domain of the register. A SWMR regular register is a bit less erratic in the sense that a read operation overlapping with write operations will at least return any of the values actually being written. Raynal presented a straightforward generalisation of the notion of SWMR safe register to the multi-writer case [27]. How the notion of SWMR regular register should be generalised to the multi-writer case, however, is less obvious. Shao et al. discuss four possibilities [28].

The formalisms in [20, 27, 28] for studying the behaviour of non-atomic registers are not directly amenable for analysing the correctness of distributed algorithms by explicit-state model checking, e.g., using the mCRL2 toolset [7]. In fact, it is not clear whether the four variants of MWMR regular registers presented in [28] will lead to a finite-state model even if the number of readers and writers and the set of data values of the register are finite. In [22], Lamport demonstrates a method of modelling SWMR safe registers through repeatedly writing arbitrary values before settling on the desired value, but this approach does not generalise to multi-writer registers. The main contribution of this paper is to present process-algebraic models of multi-writer multi-reader safe, regular and also atomic registers that can be directly used in mCRL2 to analyse the correctness of distributed algorithms.

We have used our process-algebraic models to analyse to what extent various mutual exclusion algorithms are robust for relaxed non-atomicity requirements. We find that Peterson's algorithm [26] no longer guarantees mutual exclusion if the atomicity requirement is relaxed for the turn register. A variant of Peterson's algorithm presented in [4] does guarantee mutual exclusion even if registers are only safe. The variant presented in [28], however, does not guarantee mutual exclusion with regular registers, despite a claim that it does. We also find that some of the algorithms proposed in [29, 30] do not guarantee mutual exclusion for regular registers, which seems to contradict claims that they are immune to the problem of flickering bits during writes. When analysing Lamport's 3-bit algorithm [18] we discovered that its mutual exclusion guarantee crucially depends on how one of the more complex statements of the algorithm is implemented. Finally, we confirm that Aravind's BLRU algorithm [3], Dekker's algorithm [1], Dijkstra's algorithm [9] and Knuth's algorithm [16] guarantee mutual exclusion even with safe registers.

This paper is organised as follows. In Section 2 we present some basic definitions pertaining to SWMR registers, including formalisations of Lamport's notions of SWMR safe, regular and atomic registers. In Section 3 we present and discuss our process-algebraic definitions of MWMR safe, regular and atomic registers, and establish formal relationships with their SWMR counterparts. In Section 4 we compare our notion of MWMR regular register with the variants of MWMR regular registers proposed by [28]. In Section 5 we report on our analyses of the various mutual exclusion algorithms. Finally, we present conclusions and some ideas for future work in Section 6.

2 Single-writer multi-reader registers

The definitions presented in this section are adapted from [28] and [21].

We consider n threads operating on a register with values in a finite set $\mathbb{D}$ of register values; the initial value of the register will be denoted by d_{init}. Threads are identified by a natural number in the set $\mathbb{T} = \{0, \ldots, n-1\}$. A *read operation* by thread $i \in \mathbb{T}$ on the register, with *return value* $d \in \mathbb{D}$, is a sequence $r_i(d) = sr_i fr_i(d)$ consisting of an *invocation* sr_i (for "thread i starts to read"), and a matching *response* $fr_i(d)$ (for: "the read by thread i finishes with return value d"). A *write* operation of thread i on the register, with *write value*

d, is a sequence $w_i(d) = sw_i(d)fw_i$ consisting of an *invocation* $sw_i(d)$ (for: "thread i starts to write value d") and a matching *response* fw_i (for: "the write by thread i finishes"). An *operation* of thread i is either a read operation or a write operation of that thread.

For every $i \in \mathbb{T}$, let $A_i = \{sr_i, fr_i(d), sw_i(v), fw_i \mid d \in \mathbb{D}\}$, and let $A = \bigcup_{i\in\mathbb{T}} A_i$. If σ is a sequence of elements of A, then we denote by $\sigma|i$ the subsequence of σ consisting of the elements in A_i. A *schedule* on a register is a finite or infinite sequence σ of elements of A such that $\sigma|i$ consists of alternating invocations and matching responses, beginning with an invocation, and if $\sigma|i$ is finite, ending with a response. Note that, by these requirements and our definition of the notion of operation, $\sigma|i$ can then be obtained as the concatenation of read and write operations $o_0o_1o_2\ldots$ executed by thread i.[1] We shall denote by $ops(\sigma, i)$ the set of all operations executed by thread i (i.e., $ops(\sigma, i) = \{o_0, o_1, o_2, \ldots\}$) and by $ops(\sigma)$ the set of all operations executed by any of the threads. It is technically convenient to include in $ops(\sigma)$ a special write operation w_{init}that writes the initial value of the register. Then $ops(\sigma) = \{w_{init}\} \cup \bigcup_{i\in\mathbb{T}} ops(\sigma, i)$. We also use $reads(\sigma)$ and $writes(\sigma)$ for the subsets of $ops(\sigma)$ respectively consisting of the read operations and the write operations only.

A schedule σ induces a partial order on $ops(\sigma)$: if $o, o' \in ops(\sigma)$, then we write $o <_\sigma o'$ if, and only if, the response of o precedes the invocation of o' in σ. We stipulate that $w_{init} < o$ for all $o \in ops(\sigma)\backslash\{w_{init}\}$. Let $r \in ops(\sigma)$ be a read operation and let $w \in ops(\sigma)$ be a write operation. We say that w is *fixed* for r if $w <_\sigma r$; *fix-writes*(σ, r) denotes the set of all writes that are fixed for r. We say that w is *relevant* for r if $r \not<_\sigma w$; *rel-writes*(σ, r) denotes the set of all writes in $ops(\sigma)$ that are relevant for r. Note that, by the inclusion of w_{init}, the sets *rel-writes*(σ, r) and *fix-writes*(σ, r) are non-empty for all $r \in reads(\sigma)$. We say that $r \in reads(\sigma)$ *can read from* $w \in writes(\sigma)$ if w is relevant for r and there does not exist $w' \in writes(\sigma)$ such that $w <_\sigma w' <_\sigma r$. An operation o has *overlapping writes* if there exists $w \in writes(\sigma)$ such that $o \not<_\sigma w$ and $w \not<_\sigma o$.

In [28], a register model is defined as a set of schedules satisfying certain conditions. Restricting attention to single-writer multi-reader (SWMR) registers only, Lamport considers three register models: safe, regular and atomic [21]. We proceed to define Lamport's models by formulating conditions on *single-writer* schedules, i.e., schedules in which all write operations are by one particular thread. If σ is a single-writer schedule, then, since a write cannot have overlapping writes, every non-empty finite set W of writes has a $<_\sigma$-maximum, i.e., an element $w \in W$ such that $w' <_\sigma w$ for all $w' \in W \setminus \{w'\}$. Since writes that are fixed for r have their responses in the finite prefix of σ preceding the invocation of r, we have that *fix-writes*(σ, r) is finite for every r. Since *fix-writes*(σ, r) is non-empty, it always has a $<_\sigma$-maximum.

A SWMR register is *safe* if a read that does not have overlapping writes returns the most recently written value. A read that does have overlapping writes may return any arbitrary value in the domain $\mathbb{D}$ of the register.

▶ **Definition 1.** *A single-writer schedule σ is* safe *if every read r without overlapping writes returns the value written by the $<_\sigma$-maximum of the set of fix-writes(σ, r).*

A SWMR register is *regular* if it is safe, and a read that has overlapping writes returns

[1] The same operation may occur multiple times in $\sigma|i$. Henceforth, when we consider an operation in $\sigma|i$ we actually mean to refer to a specific occurrence in $\sigma|i$ of the operation. To disambiguate between two different occurrences of the same operation o we could, e.g., annotate each occurrence of o with its position in $\sigma|i$. We will not do so explicitly, because it will unnecessarily clutter the presentation. But the reader should keep in mind that, whenever we refer to an operation in a schedule σ we actually mean to refer to a particular occurrence of that operation in $\sigma|i$.

the value of one of the overlapping writes or the most recently written value.

▶ **Definition 2.** *A single-writer schedule σ is* regular *if every read r returns either the value written by the $<_\sigma$-maximum of the set fix-writes(σ, r) or the value of an overlapping write.*

A SWMR register is *atomic* if all reads and writes behave as though they occur in some definite order. A *serialisation* is a total order $\mathcal{S}$ on a subset O of $ops(\sigma)$ that is *consistent* with $<_\sigma$ in the sense that for all $o, o' \in O$ we have that $o <_\sigma o'$ implies $o \,\mathcal{S}\, o'$. A serialisation $(O, \mathcal{S})$ is *legal* if every read operation returns the value of the most recent write operation according to $\mathcal{S}$, that is, whenever $r \in O$ is a read operation with return value v, then v is the write value of $\mathcal{S}$-maximum of $rel\text{-}writes(\sigma, r)$.

▶ **Definition 3.** *A single-writer schedule σ is* atomic *if $ops(\sigma)$ has a legal serialisation.*

3 Multi-writer multi-reader registers

We now want to define multi-write multi-reader (MWMR) safe, regular and atomic registers. Since our goal is to verify the correctness of mutual exclusion algorithms by model checking, we prefer operational, process-algebraic definitions of register models over definitions in terms of schedules. We are going to define register models by giving recursive process definitions that, given the state of the register, admit certain interactions with the register, resulting in an update of the state of the register. Which information needs to be maintained in the state of the register depends on the register model, but the state of register should at least reflect which operations are currently active. So, with each register model $m \in \{s, r, a\}$ we associate a set of states $\mathbb{S}_m$, and we assume that the following functions are defined on $\mathbb{S}_m$:

$$\begin{aligned} &rdrs, wrtrs, idle : \mathbb{S}_m \to \mathcal{P}(\mathbb{T}) \\ &usr_i, ufr_i, ufw_i : \mathbb{S}_m \to \mathbb{S}_m \\ &usw_i : \mathbb{D} \times \mathbb{S}_m \to \mathbb{S}_m \ . \end{aligned} \tag{1}$$

The mappings $rdrs$ returns the set of all threads that are currently reading, i.e., $i \in rdrs(s)$ if, and only if, thread i has invoked a read operation but the matching response has not yet occurred. Similarly, $wrtrs$ returns the set of all threads that are currently writing, and $idle$ returns the set of all threads that are currently not reading and not writing. The mappings usr_i, ufr_i, usw_i and ufw_i perform update operations on the state of the register, corresponding to whether the most recent interaction of the register was an invocation (usr_i) or response (ufr_i) of a read, or an invocation (usw_i) or a response (ufw_i) of a write. The update operation usw_i also takes the write value into account.

In the remainder of this section we shall first present our models of MWMR safe, regular and atomic registers, and then comment on the representation of these models in mCRL2.

3.1 MWMR Safe Registers

Lamport's SWMR safe register model (see Definition 1) accounts for how reads and writes behave when they do not have overlapping writes, and how reads behave when they do have overlapping writes. To generalise Lamport's notion to MWMR registers, we need to define how writes behave when they have overlapping writes. We follow Raynal's approach and define that when a write has overlapping writes, then its effect is that some arbitrary value in $\mathbb{D}$ is written to the register [27].

Our process-algebraic definition of a MWMR safe register is shown in Figure 1. The equation defines the behaviour of processes $R_s(d, s)$; the parameter $d \in \mathbb{D}$ reflects the current

$$R_s(d:\mathbb{D}, s:\mathbb{S}_s) = \sum_{i\in\mathbb{T}} \left(\begin{array}{ll} & (i \in \mathit{idle}(s)) \to \mathit{sr}_i \cdot R_s(d, \mathit{usr}_i(s)) \\ + & (i \in \mathit{idle}(s)) \to \sum_{d'\in\mathbb{D}} \mathit{sw}_i(d') \cdot R_s(d, \mathit{usw}_i(d', s)) \\ + & (i \in \mathit{rdrs}(s) \wedge \neg \mathit{overlap}_i(s)) \to \mathit{fr}_i(d) \cdot R_s(d, \mathit{ufr}_i(s)) \\ + & (i \in \mathit{rdrs}(s) \wedge \mathit{overlap}_i(s)) \to \sum_{d'\in\mathbb{D}} \mathit{fr}_i(d') \cdot R_s(d, \mathit{ufr}_i(s)) \\ + & (i \in \mathit{wrtrs}(s) \wedge \neg \mathit{overlap}_i(s)) \to \mathit{fw}_i \cdot R_s(\mathit{next}(s), \mathit{ufw}_i(s)) \\ + & (i \in \mathit{wrtrs}(s) \wedge \mathit{overlap}_i(s)) \to \sum_{d'\in\mathbb{D}} \mathit{fw}_i \cdot R_s(d', \mathit{ufw}_i(s)) \end{array} \right)$$

Figure 1 Safe register model

value of the register, and the parameter $s \in \mathbb{S}_s$ reflects its current state. For the behaviour of the safe register it must be determined for every read or write operation of a thread whether, during its interaction with the register, there was an overlapping write operation by some other thread. Therefore, in addition to the functions specified in Equation 1, we presuppose on $\mathbb{S}_s$ a predicate $\mathit{overlap}_i$ such that $\mathit{overlap}_i(s)$ holds if during the interaction of thread i with the register there was an overlapping write by another thread. At the response of a write that is not overlapping with other writes, the current value d of the register needs to be replaced by the write value. Hence, whenever a write is invoked, the write value is stored in s through $\mathit{usw}_i(s)$; this value can be retrieved with the mapping $\mathit{next} : \mathbb{S}_s \to \mathbb{D}$ if the write had no overlapping writes. If there were overlapping writes, next is undefined. The right-hand side of the equation in Figure 1 specifies the behaviour of the register using standard process-algebraic operations: $\cdot$ denotes sequential composition, $+$ denotes non-deterministic choice, $\to$ denotes a conditional, and $\sum$ denotes choice quantification [13].

The definition in Figure 1 induces a transition relations $\xrightarrow{a}$ $(a \in A)$ on the set of tuples $\langle d, s\rangle$ $(d \in \mathbb{D},\ s \in \mathbb{S}_s)$. For instance, if $i \in \mathit{rdrs}(s)$ and $\neg\mathit{overlap}_i(s)$, then there is a transition

$$\langle d, s\rangle \xrightarrow{\mathit{fr}_i(d)} \langle d, \mathit{ufr}_i(s)\rangle \ ,$$

according to the third summand of the definition in Figure 1; and if $i \in \mathit{wrtrs}(s)$ and $\mathit{overlap}_i(s)$, then, for every $d' \in \mathbb{D}$, there is a transition

$$\langle d, s\rangle \xrightarrow{\mathit{fw}_i} \langle d', \mathit{usw}_i(s)\rangle \ ,$$

according to the last summand of the definition in Figure 1.

We let s_{init} denote the *initial state* of the safe register, and we define $\mathit{idle}(s_{\mathit{init}}) = \mathbb{T}$, $\mathit{wrtrs}(s_{\mathit{init}}) = \mathit{rdrs}(s_{\mathit{init}}) = \emptyset$, $\mathit{overlap}_i(s)$ is false, and $\mathit{next}(s) = d_{\mathit{init}}$. Henceforth, we shall abbreviate $R_s(d_{\mathit{init}}, s_{\mathit{init}})$ by R_s. A *trace* of R_s is a finite or infinite sequence $a_0a_1\cdots a_{n-1}a_n\cdots$ of elements of A such that there exist $d_0, d_1, d_2, \ldots, d_n, \ldots \in \mathbb{D}$ and $s_0, s_1, s_2, \ldots, s_n, \ldots \in \mathbb{S}_s$ with $d_0 = d_{\mathit{init}}$ and $s_0 = s_{\mathit{init}}$ and $\langle d_0, s_0\rangle \xrightarrow{a_0} \langle d_1, s_1\rangle \xrightarrow{a_1} \cdots \xrightarrow{a_{n-1}} \langle d_n, s_n\rangle \xrightarrow{a_n} \cdots$. We denote by $\mathcal{T}_s$ the set of all traces of R_s. A trace $\alpha \in \mathcal{T}_s$ is *complete* if, for all $i \in \mathbb{T}$, either $\alpha|i$ is infinite or $\alpha|i$ ends with a response. A *single-writer* trace is a trace in which all invocations and responses of write operations are by the same thread.

We argue that there is a one-to-one correspondence between the single-writer safe schedules and the single-writer complete traces of R_s. First, note that schedules and complete traces adhere to exactly the same restrictions regarding the order in which invocations and responses of read and write operations can occur: the invocation of an operation by some thread can only occur when that same thread is not currently executing another operation, and

a response to some thread for an operation can only occur if the last interaction of that thread was, indeed, an invocation of that same operation. Write values are not restricted in schedules, nor in complete traces. Moreover, in the single-writer case the value of the parameter d of the process R_s will always be the write value of write operation of which the execution finished last. Finally, note that both in schedules and in complete traces of R_s, if a read operation overlaps with a write operation, then it may return any value, and if it does not, then it will, indeed, return the value of the most recent write operation.

▶ **Proposition 4.** *Every single-writer safe schedule is a trace of R_s, and every complete single-writer trace of R_s is a safe schedule.*

3.2 MWMR regular registers

According to Lamport's definition of SWMR regular registers (see Definition 2), a read either returns the write value of the $<_\sigma$-maximum of $\mathit{fix\text{-}writes}(\sigma, r)$ or the value written by one of its overlapping writes. When writes may have overlapping writes, then $\mathit{fix\text{-}writes}(\sigma, r)$ may not have a $<_\sigma$-maximum. It is then necessary to determine, for every read r, which of the $<_\sigma$-maximal elements of $\mathit{fix\text{-}writes}(\sigma, r)$ should be taken into account when determining the return value of r, and to what extent different reads should agree on this choice.

Our considerations are as follows. First, we want our MWMR regular register model to coincide with Lamport's SWMR regular register model when there are no writes overlapping other writes, so that our analyses of algorithms that rely on SWMR regular registers are valid with respect to Lamport's model. Second, our model should be suitable for explicit-state model checking. This precludes any definition that requires keeping track of unbounded information pertaining to the history of the computation. To limit the amount of information that the model is required to remember, we let the register commit to a unique value when there are no active writes. In this respect, our model deviates from three of the four models considered in [28]; in Section 4 we provide a more detailed comparison.

To be consistent with Lamport's SWMR regular registers, a read r should be able return the value of any *overlapping* write. To determine which of the elements of the *fixed* writes is taken into account when determining the return value of r, our model non-deterministically inserts a special *order* action ow_i somewhere between the invocation and the response of every write of every thread $i \in \mathbb{T}$. One may think of the order action as marking the moment at which the write truly takes place. Note that this order action is purely for modelling purposes, we make no claims on the implementation of a regular register. The write value associated with the most recent order action preceding the invocation of a read (or the initial value if no order actions have occurred yet) is taken into account as possible return value for that read. Thus, a serialisation of all writes is generated on-the-fly through the order actions: all read operations agree on the order of the writes.

$$R_r(d:\mathbb{D}, s:\mathbb{S}_r) = \sum_{i\in\mathbb{T}} \left(\begin{array}{ll} & (i \in \mathit{idle}(s)) \to \mathit{sr}_i \cdot R_r(d, \mathit{usr}_i(s)) \\ + & (i \in \mathit{idle}(s)) \to \sum_{d'\in\mathbb{D}} \mathit{sw}_i(d') \cdot R_r(d, \mathit{usw}_i(d', s)) \\ + & (i \in \mathit{rdrs}(s)) \to \sum_{d'\in \mathit{pval}_i(s)} \mathit{fr}_i(d') \cdot R_r(d, \mathit{ufr}_i(s)) \\ + & (i \in \mathit{pndng}(s)) \to \mathit{ow}_i \cdot R_r(\mathit{wval}_i(s), \mathit{uow}_i(s)) \\ + & (i \in \mathit{wrtrs}(s) \wedge i \notin \mathit{pndng}(s)) \to \mathit{fw}_i \cdot R_r(d, \mathit{ufw}_i(s)) \end{array} \right)$$

Figure 2 Regular register model

Our process-algebraic definition of a MWMR regular register is given in Figure 2. Here, $\mathbb{S}_r$ denotes the set of possible states of the MWMR regular register. The register keeps track of the readers, writers and idle threads, similar to the safe register. It additionally keeps track of the set $pndng(s)$ of threads that have invoked a write but for which the order action has not yet occurred. The update function $uow_i : \mathbb{S}_r \to \mathbb{S}_r$ associated with the order action ow_i removes thread i from $pndng(s)$. For every thread $i \in pndng(s)$, $wval_i(s)$ is the write value of that write; it is used to correctly update the current value d of the register when ow_i occurs. For every thread $i \in rdrs(s)$, $pval_i(s)$ is the set of values that a read r invoked by thread i may return. That is, it consist of the values of all writes overlapping with r (thus far) and the value of the write with the most recent ow_j before the invocation of r.

For $i \in \mathbb{T}$, let $A_i^r = A_i \cup \{ow_i\}$, and let $A^r = \bigcup_{i\in\mathbb{T}} A_i^r$. The process definition in Figure 2 induces transition relations $\xrightarrow{a}$ $(a \in A^r)$ on the set of tuples $\langle d, s\rangle$ $(d \in \mathbb{D},\ s \in \mathbb{S}_r)$. As before $idle(s_{init}) = \mathbb{T}$, $rdrs(s_{init}) = wrtrs(s_{init}) = \emptyset$. We also have $pndng(s_{init}) = \emptyset$, and $pval_i(s_{init}) = \emptyset$ for all $i \in \mathbb{T}$. The initial values for $wval_i(s_{init})$ do not matter, since $wval_i(s)$ only matters when $i \in pndng(s)$. We use R_r to abbreviate $R_r(d_{init}, s_{init})$, and define a trace of R_r, also as before, as a finite or infinite sequence of elements of A^r appearing as labels in a transition sequence starting at $\langle d_{init}, s_{init}\rangle$. We denote by $\mathcal{T}_r$ the set of all traces of R_r.

Compared to schedules, the traces of R_r have extra ow_i actions. If α is a finite or infinite sequence of elements of A^r, then we denote by $\bar{\alpha}$ the sequence of elements of A obtained from α by deleting all occurrences of ow_i $(i \in \mathbb{T})$. We can then formulate a correspondence between the single-writer traces of R_r (i.e., the traces in which all invocations and responses of write operations are by the same thread) and single-writer regular schedules.

If writes have no overlapping writes, then the most recent order action when a read r is invoked either corresponds to the $<_\sigma$-maximum of $\mathit{fix\text{-}writes}(\sigma, r)$, or to a write that overlaps with r. In the first case, the set of possible values that can be returned by the read according to our model will coincide with the set of possible values that it can return according to Definition 2. In the latter case, our model allows a subset of the values possible according to Definition 2 to be returned. Hence, a read in our model never returns a value that could not be returned according to Lamport's SWMR definition of regular registers. Moreover, if there is a trace of R_r in which the order action ow_i of a write that overlaps with r occurs before the invocation of r, then there also exist a trace in which it occurs after the invocation of r. Thus, the set of traces described by our model includes all regular schedules according to Definition 2 whenever there are no writes overlapping other writes.

▶ **Proposition 5.** *For every single-writer regular schedule σ there is a trace α of R_r such that $\bar{\alpha} = \sigma$, and if α is a complete single-writer trace of R_r, then $\bar{\alpha}$ is a regular schedule.*

3.3 MWMR atomic registers

Definition 3, formalising Lamport's notion of SWMR atomic register, straightforwardly generalises to MWMR registers by omitting the single-writer restriction on schedules. Our process-algebraic model should generate the legal serialisation of all operations on-the-fly. To this end, we introduce, for every thread i, *execution* actions er_i and ew_i to mark the exact moment at which an operation is treated as occurring. An operation's execution action must, of course, occur between its invocation and response. The value that is returned at the response of a read is the value that the register stored at the moment of that read's execution; the register's stored value is updated to a write's value at that write's execution.

The process-algebraic model of our MWMR atomic register is shown in Figure 3. The set of states of R_a is denoted by $\mathbb{S}_a$. In addition to the standard update functions, there are

extra update functions $uer_i, uew_i : \mathbb{S}_a \to \mathbb{S}_a$ for the execution actions. The effect of applying uer_i on s is to store the current value d of the register as the value that should be returned at the response of the active read by thread i; this value can then be retrieved with $vals_i(s)$, and $vals_i(s) = \bot$ until then. The effect of applying uew_i is to update the current value d of the register to the write value of the active write by thread i; this value can also be retrieved with $vals_i(s)$, and $vals_i(s) = \bot$ thereafter. Note that, by setting $vals_i(s)$ to $\bot$ before a read has been executed and after a write has been executed, we can use $vals_i(s)$ in combination with $rdrs(s)$ and $wrtrs(s)$ to determine whether the execution of an operation has taken place.

$$R_a(d : \mathbb{D}, s : \mathbb{S}_a) = \sum_{i \in \mathbb{T}} \left(\begin{array}{ll} & (i \in idle(s)) \to sr_i \cdot R_a(d, usr_i(s)) \\ + & (i \in idle(s)) \to \sum_{d' \in \mathbb{D}} sw_i(d') \cdot R_a(d, usw_i(d', s)) \\ + & (i \in rdrs(s) \wedge vals_i(s) = \bot) \to er_i \cdot R_a(d, uer_i(s)) \\ + & (i \in wrtrs(s) \wedge vals_i(s) \neq \bot) \to ew_i \cdot R_a(vals_i(s), uew_i(s)) \\ + & (i \in rdrs(s) \wedge vals_i(s) \neq \bot) \to fr_i(vals_i(s)) \cdot R_a(d, ufr_i(s)) \\ + & (i \in wrtrs(s) \wedge vals_i(s) = \bot) \to fw_i \cdot R_a(d, ufw_i(s)) \end{array} \right)$$

Figure 3 Atomic register model

For $i \in \mathbb{T}$, let $A_i^a = A \cup \{er_i, ew_i\}$, and let $A^a = \bigcup_{i \in \mathbb{T}} A_i^a$. The process definition in Figure 3 induces transition relations $\xrightarrow{a}$ $(a \in A^a)$ on the set of tuples $\langle d, s \rangle$ $(d \in \mathbb{D},\ s \in \mathbb{S}_a)$. As before $idle(s_{init}) = \mathbb{T}$ and $rdrs(s_{init}) = wrtrs(s_{init}) = \emptyset$; the initial values for $vals_i(s_{init})$ do not matter. We use R_a to abbreviate $R_a(d_{init}, s_{init})$, and define a trace of R_A, also as before, as a finite or infinite sequence of elements of A^A appearing as labels in a transition sequence starting at $\langle d_{init}, s_{init} \rangle$. We denote by $\mathcal{T}_a$ the set of all traces of R_a.

Compared to schedules, the traces of R_a have extra er_i and ew_i actions. If α is a finite or infinite sequence of elements of A^a, then we denote by $\bar{\alpha}$ the sequence obtained from α by deleting all occurrences of er_i and ew_i for $i \in \mathbb{T}$. The correspondence between atomic schedules and complete traces of R_a follows straightforwardly. It suffices to prove that R_a admits exactly those traces α such that there exists a legal serialisation of $\bar{\alpha}$. To this end, note that the execute actions provide such a serialisation, and the definition of R_a has the responses of operations behave in accordance with this serialisation.

▶ **Proposition 6.** *For every atomic schedule σ there is a trace α of R_a such that $\bar{\alpha} = \sigma$, and if α is a complete trace of R_a, then $\bar{\alpha}$ is an atomic schedule.*

3.4 mCRL2 implementation

The mCRL2 toolset [7] provides tools for model checking and equivalence checking. Models are defined in the mCRL2 language [13], which comprises a process-algebraic specification language and facilitates the algebraic specification of data types. Properties defined in the modal μ-calculus can be checked on those models. One nice feature of mCRL2 is that when a property does not hold a counterexample can be generated. For more information we refer to [13] as well as the toolset's website[2].

[2] `https://www.mcrl2.org`

We have implemented the models presented in Figures 1, 2 and 3 in the mCRL2 language. By adding processes that model the threads executing the desired algorithm in a manner compatible with the interface of the register models, we can verify the same algorithm easily under different atomicity assumptions. An added benefit is that we can assume different levels of atomicity for different registers simultaneously, so that we pinpoint exactly to what extent the algorithm is robust for non-atomicity. The model can be found as part of the examples delivered with the mCRL2 distribution[3].

The mCRL2 language has support for standard data types such as sets, bags and arrays (implemented as mappings) as well an algebraic specification facility to define new datatypes. This allows us to model the register models staying close to the process-algebraic models presented in this paper.

4 Alternative definitions of MWMR regular registers

In [28] four definitions for MWMR regular registers are proposed. These are formulated as conditions on schedules. We discuss how our definition of MWMR regular registers relates to these definitions.

The following definition captures the weakest condition on schedules presented in [28].

▶ **Definition 7.** *A schedule σ satisfies the* weak *condition if, for every read operation r in $ops(\sigma)$, there exists a legal serialisation of $writes(\sigma) \cup \{r\}$.*

It follows straightforwardly from our MWMR regular register definition that any complete trace $\alpha \in \mathcal{T}_r$, when transformed into a schedule $\bar{\alpha}$ by deleting the order actions, satisfies Definition 7. As explained in Section 3.2, our model generates a serialisation of all writes. For every read r by thread i, it returns either the value of the last write in this serialisation before sr_i, or the value of one of the writes overlapping this read. In both cases, we may obtain a legal serialisation of $writes(\bar{\alpha}) \cup \{r\}$ by taking the serialisation of writes associated with $\bar{\alpha}$ and inserting r right after the write that it reads from. This is consistent with $<_\sigma$ because the serialisation of the writes is, and r will only be placed after a write that either has its response before the invocation of r, or that r overlaps with.

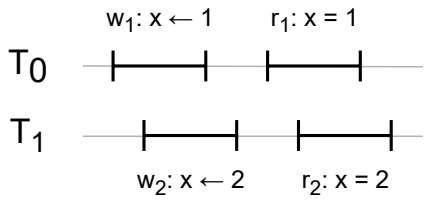
▶ **Proposition 8.** *If $\alpha \in \mathcal{T}_r$ is complete, then the schedule $\bar{\alpha}$ satisfies the weak condition.*

In all our MWMR register definitions it is the case that when no writes are active on a register, it stores a unique value. It reduces the burden of storing elaborate information on the execution history of the register, as would be necessary with the definitions of [28], and thus leads to a smaller statespace. A consequence of our choice is that not all schedules satisfying the weak condition can be generated by our model.

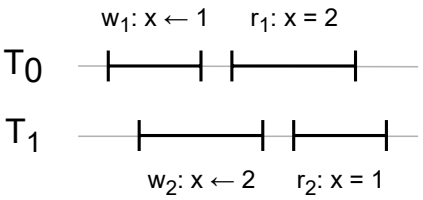
▶ **Example 9.** Consider the schedule depicted in Figure 4a. It is argued in [28, Figure 6] that it satisfies the weak condition, but it cannot be generated by our regular register model R_r because once w_1 and w_2 have ended, the register will have stored a unique value (either 1 or 2). Hence, the return values of r_1 and r_2 cannot be different. Note that, for the same reason, the schedule cannot be generated by our safe register model R_s.

As illustrated in the preceding example, there exist schedules satisfying the weak condition that cannot be generated by our safe register model R_s. Conversely, it is easy to see that

[3] `https://github.com/mCRL2org/mCRL2/tree/master/examples/academic/non-atomic_registers` (`972629b`)



(a) A schedule allowed by the weak, reads-from and no-inversion definitions but not by our regular register model.



(b) A schedule allowed by our regular register model but not by the write-order definition.

Figure 4 Schedules demonstrating the differences between our regular register model and the definitions in [28]. We illustrate these schedules on a timeline, where an operation is drawn from its invocation to its response.

there exist complete traces generated by our safe register model R_s (e.g., with overlapping writes resulting in a value that is not written by any of the writes) that do not satisfy the weak condition.

The second condition in [28] associates with every read operation a serialisation and formulates a consistency requirement on these serialisations. If $r \in \mathit{reads}(\sigma)$, then an *$r$-serialisation* is a serialisation $\mathcal{S}_r$ on $\mathit{rel\text{-}writes}(\sigma) \cup \{r\}$.[4]

▶ **Definition 10.** *A schedule σ satisfies* write-order *if for each read r in $\mathit{ops}(\sigma)$ there exists a legal serialisation $\mathcal{S}_r$ of $\mathit{rel\text{-}writes}(\sigma) \cup \{r\}$ satisfying the following condition: for all reads r_1, r_2 in $\mathit{ops}(\sigma)$, and for all writes $w_1, w_2 \in \mathit{rel\text{-}writes}(\sigma, r_1) \cap \mathit{rel\text{-}writes}(\sigma, r_2)$ it holds that $w_1 \; \mathcal{S}_{r_1} \; w_2$ if and only if $w_1 \; \mathcal{S}_{r_2} \; w_2$.*

▶ **Proposition 11.** *For every schedule σ satisfying the write-order condition, there exists a trace α in $\mathcal{T}_r$ such that $\bar{\alpha} = \sigma$.*

We give a brief, informal description of how such a trace α can be constructed here; a more formal argument is presented in Appendix A. The idea is that order actions can be inserted between the invocation and response of every write in σ, such that the return values of the reads match this placement of order actions. Note that for reads that return the value of an overlapping write, this return value is possible according to Figure 2 regardless of how the order actions are placed. In our placement of order actions, we therefore only need to carefully consider reads that return the value of a write that is fixed for them. According to Definition 10, reads in σ agree on the relative ordering of all writes that are relevant to them. Since $\mathit{fix\text{-}writes}(\sigma, r) \subseteq \mathit{rel\text{-}writes}(\sigma, r)$ for every read r, the reads also agree on the relative ordering of the fixed writes. We use this information to construct an ordering on all writes that is consistent both with $<_\sigma$ and with the return values of reads that read from writes that are fixed for them. Effectively, we find a single view on the relative order of all the write operations that is possible for every read in the schedule that returns the value of a fixed write. Using this ordering, we can then place the order actions in the schedule σ to create the trace $\alpha \in \mathcal{T}_r$ such that $\bar{\alpha} = \sigma$.

[4] By considering serialisations of the relevant writes for r, instead of all writes, we deviate from [28]. Since a serialisation $\mathcal{S}$ on $\mathit{writes}(\sigma) \cup \{r\}$ must be consistent with $<_\sigma$, we will have that $r \; \mathcal{S} \; w$ for all $w \in \mathit{writes}(\sigma) \setminus \mathit{rel\text{-}writes}(\sigma)$. It follows that the restriction of a serialisation $\mathcal{S}$ on $\mathit{writes}(\sigma) \cup \{r\}$ to $\mathit{rel\text{-}writes}(\sigma) \cup \{r\}$ is an r-serialisation, and $\mathcal{S}$ is legal if, and only if, its restriction is.

Whilst every schedule satisfying Definition 10 corresponds to a trace of our model, not every schedule with a corresponding trace in our model is allowed by the write-order condition.

▶ **Example 12.** Consider Figure 4b. This schedule is allowed by our model; r_1 can read 2 in x because it overlaps with w_2 and it is possible for r_2 to read 1 if the order action of w_2 is done before the order action of w_1. This schedule does not meet Definition 10 however; since both writes to x are relevant for both reads, the two reads must agree on the respective order of the writes. For r_2 to read 1, it must be the case that $w_2 \mathcal{S}_{r_2} w_1$. But since $w_1 < r_1$ according to the schedule, this means that $w_2 \mathcal{S}_{r_1} w_1 \mathcal{S}_{r_1} r_1$, so r_1 cannot read 2.

The third and fourth conditions on schedules proposed in [28] we refer to as *reads-from* [28, Definition 9] and *no-inversion* [28, Definition 10], respectively. We do not recall these conditions here, and instead refer to [28] for more details.

Our notion of MWMR regular register is incomparable with the notions induced by the reads-from and no-inversion conditions on schedules. First, as already indicated, every schedule that satisfies the write-order condition is also allowed by our model. As it is proven in [28] that the write-order condition is incomparable with the reads-from and no-inversion conditions, this means our model admits schedules not admitted by these definitions. To see that that not all schedules satisfying reads-from and no-inversion are admitted by our model, it suffices to observe that the schedule presented in Figure 4a, which is not admitted by our MWMR regular register model, satisfies the reads-from and the no-inversion conditions. (See, e.g., [28, Figure 8] and [28, Figure 9], which satisfy the reads-from and no-inversion conditions, respectively, and have the schedule in Figure 4a as prefix.)

5 Verifying Mutual Exclusion Protocols

We have used the register processes described in Section 3 to analyse several well-known mutual exclusion algorithms. To this end, we have modelled the behaviour of the threads as prescribed by the algorithm also as processes, which interact with the register processes. That a thread is executing its non-critical section is represented in our model by the action *noncrit*, and that is executing its critical section is represented by the action *crit*; both actions are parameterised with the thread id. We have checked the following two properties.

▶ Property 1 (Mutex). There is no state reachable from the initial state of the model in which there are two distinct threads i and j such that $crit(i)$ and $crit(j)$ are both enabled in this state.

▶ Property 2 (Reach). For all threads i, always after an occurrence of a $noncrit(i)$ action it holds that, as long as a $crit(i)$ action has not happened, a state is reachable in which $crit(i)$ is enabled.

The Reach property is implied by starvation freedom, and so if it does not hold, then neither does starvation freedom. We chose to analyse this property rather than starvation freedom itself because the presence of busy waiting loops in our models would require us to use fairness assumptions to dismiss spurious counterexamples. The question of how to interpret fairness assumptions when dealing with non-atomic registers is outside of the scope of this paper.

The results of our verification are shown in Table 1. When doing model checking, we have to instantiate a specific number of threads. We have restricted our verification to three threads for all algorithms, except for Dekker, Attiya-Welch and Peterson, which are only defined for two threads.

	Safe		Regular		Atomic	
	Mutex	Reach	Mutex	Reach	Mutex	Reach
Aravind (BLRU) [3, Figure 4]	✓	✓	✓	✓	✓	✓
Attiya-Welch [4, Algorithm 12]	✓	✓	✓	✓	✓	✓
Attiya-Welch alternate [28, Figure 19.1]	✓	×	✓	×	✓	✓
Dekker [1, Figure 1]	✓	✓	✓	✓	✓	✓
Dijkstra [9]	✓	✓	✓	✓	✓	✓
Knuth [16]	✓	✓	✓	✓	✓	✓
Lamport (3-bit) [18, Figure 2]	✓	✓	✓	✓	✓	✓
Peterson [26]	×	✓	×	✓	✓	✓
Szymanski (flag) [29, Figure 2]	×	×	×	✓	✓	✓
Szymanski (flag with bits)	×	✓	×	✓	×	✓
Szymanski (3-bit lin. wait) [30, Figure 1]	×	✓	×	✓	×	✓

Table 1 Results of verifying mutual exclusion algorithms.

In this section, we discuss some of our most interesting findings. For complete descriptions of counterexamples, as well as further discussion of our results we refer to Appendix B. All models are available through GitHub[5].

5.1 Peterson's Algorithm

Peterson's classic algorithm (see Algorithm 1) was not designed to be correct under non-atomic register assumptions. An analysis of the mutual exclusion violation with safe registers still gives interesting insights into the algorithm and some of the unexpected behaviour of safe registers.

As expected, mCRL2 reports that mutual exclusion does not hold when using non-atomic registers. We present a visualisation of the counterexample generated by mCRL2 for safe registers in Figure 5. There are two instances of overlapping operations. First, since the two writes to *turn*, labelled w_3 and w_4 in Figure 5, overlap, according to the safe register model the register can have any arbitrary value after they both have ended. In this counterexample, *turn* has the value 1, which allows thread 0 to read the value 1 (the read labelled r_4) and enter the critical section. Second, thread 1's read of *turn* (labelled r_2) overlaps with thread 0's write (labelled w_4). The read can therefore return an arbitrary value, in this case the value 0, which allows thread 1 to enter the critical section.

[5] `https://github.com/mCRL2org/mCRL2/tree/master/examples/academic/non-atomic_registers` (`972629b`)

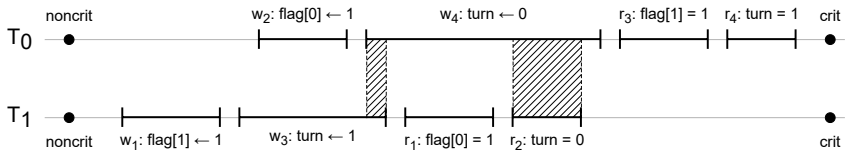


Figure 5 Counterexample generated by mCRL2 for mutual exclusion for Peterson's algorithm with safe registers, represented on a timeline

This counterexample shows only overlaps on the *turn* register. We can initialise our model such that the *turn* register is atomic, but both *flag* registers behave as safe registers. We find that mutual exclusion does hold then. This confirms that overlapping operations on the *turn* register are the sole cause of the mutual exclusion violation for Peterson's algorithm. We discuss Peterson's algorithm with regular registers in Appendix B.

Algorithm 1 Peterson's algorithm for two threads from [26]. We use i for the thread's own id and j for the other thread's id.

1: $\mathit{flag}[i] \leftarrow 1$
2: $\mathit{turn} \leftarrow i$
3: **await** $\mathit{flag}[j] = 0 \vee \mathit{turn} = j$
4: **critical section**
5: $\mathit{flag}[i] \leftarrow 0$

Algorithm 2 Szymanski's flag algorithm from [29], i is the thread's own id.

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

5.2 Szymanski's Flag Algorithm

There are several variants of Szymanski's algorithm, which all seem to have been derived from the flag-based algorithm shown as Algorithm 2. In [29], Szymanski proposes this flag-based algorithm and claims that an implementation of it representing the flags using three bits is robust for flickering of bits (i.e., is correct for non-atomic registers). As indicated in Table 1, we find that neither the integer nor the bits variant ensure mutual exclusion when using non-atomic registers. The full analysis of the bits version, as well as a variant of it known as the 3-bit linear wait algorithm [30] are presented in Appendix B. Here, we only discuss the integer version of the flag algorithm, as the counterexample against Mutex that we have found illustrates the core issue shared by all mentioned variants of Szymanski's algorithm.

The pseudocode for the flag algorithm is shown in Algorithm 2. It is originally presented in [29, Figure 2], but note that we have repaired an obvious typo: [29, Figure 2] erroneously has a conjunction instead of a disjunction in line 10. All *flag* registers are initialised at 0.

See Figure 6 for a visualisation of the counterexample for mutual exclusion with two threads and regular registers that we found using the mCRL2 toolset. The first instance of a read overlapping with a write is irrelevant, reading $\mathit{flag}[1] = 1$ would also have been possible without overlap. The other two instances of overlap are of interest. Thread 0 is writing the value 3 to $\mathit{flag}[0]$ and thread 1 reads $\mathit{flag}[0]$ twice while this write is active. The first time it reads the new value (3), while the second time it reads the old value (1). Lamport specifically highlights that such a sequence is possible when using regular registers [21]. Since only single-writer registers are used and write-order reduces to Lamport's definition of regular registers when single-writers in that case [28], this counterexample is also valid for write-order.

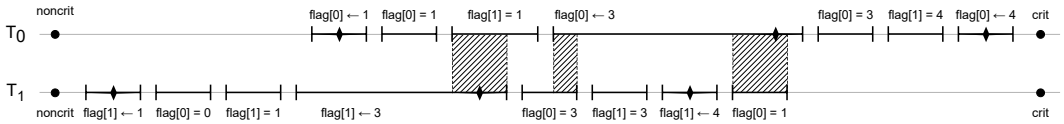


Figure 6 Mutex violation for Szymanski (flag) with regular registers and two threads, generated by mCRL2, on a timeline. The order-actions are drawn with lines during a write's execution.

5.3 Implementation Details

Our analyses have also revealed that seemingly minor implementation subtleties can make the difference between a correct and an incorrect algorithm. A non-atomic register that is read multiple times in a row may return different values, even if no new writes to this register have started. This means that when the value of a register needs to be checked several times in an algorithm, there is a difference between reading it once and subsequently checking a local copy of the value, or reading it again when needed.

For an example where this affects correctness, consider the Attiya-Welch algorithm. While the presentation in [4, p. 77] ensures reachability of the critical section with safe registers, the seemingly equivalent reformulation of this same algorithm in [28] does not. The latter suggests that a thread needs to read a particular register twice as part of two different conditions that in the former are handled simultaneously. In [28], that presentation of the algorithm is claimed to be correct under all four of their MWMR regular register models; our counterexample shows that it is not. A similar phenomenon occurs with Lamport's 3-bit algorithm, in which each thread i has a bit z_i. As part of the algorithm, a computation is done on z (the function assigning z_i to i). Lamport states that "evaluating $[z]$ at j requires a read of the variable z_j." This may lead one to implement this algorithm by having threads re-read variables whenever needed. It turns out this implementation leads to a deadlock. Locally saving all required z-values at the start of the computation and then only referencing this local copy during the computation solves this issue. Consequently, these algorithms have a correct implementation, but they are also easily implemented incorrectly. See the discussions of Attiya-Welch and Lamport in Appendix B for more details.

5.4 Other Verifications

There have been many mechanical verifications of mutual exclusion algorithms with atomic registers. For instance, in recent tutorials on the verification of distributed algorithms in mCRL2, verifications of Dekker's and Peterson's algorithms are presented [11, 12]. Several such verifications have also been done with the CADP toolset; see, e.g., [25] for the results of verifying a large number of mutual exclusion algorithms, including Szymanski, Dekker and Peterson, with atomic registers.

To the best of our knowledge, we are the first to propose a systematic approach to mechanically verifying the correctness of mutual exclusion algorithms with respect to non-atomic registers, but there have been some mechanical verifications for specific algorithms.

Lamport himself modelled the Bakery algorithm in TLA+, representing the non-atomic writes as sequences of write actions of arbitrary length, where every action results in an arbitrary value being written, except for the last which writes the intended value [22]. This approach for modelling safe registers only works for SWMR registers; it does not work for MWMR registers. This approach for modelling safe SWMR registers, as well as a similar approach for modelling regular SWMR registers, is presented in [2]. This approach is also used

in several verifications done by Wim Hesselink, including of the Lycklama–Hadzilacos–Aravind algorithm in [15] and the Bakery algorithm in [14].

In [8], several mutual exclusion algorithms are verified with atomic registers using timed automata in UPPAAL. Additionally, the Block & Woo algorithm is checked with bit flickering. Their model does not account for writes that overlap with other writes. Additionally, their model for the behaviour of safe registers is specific to the registers used in the algorithm.

Dekker's algorithm with safe registers is considered in [6]. There it is demonstrated that Dekker's algorithm does not satisfy starvation freedom when safe registers are used, and a fixed version of the algorithm is presented.

Szymanski's flag algorithm with atomic registers is proven correct in [24]. This paper demonstrates the importance of checking all threads in the "forall" and "exists" statements in the pseudocode in the same order every time. This is also how we model the algorithm.

There have been other verifications of Szymanski's algorithms [23, 31], the former paper using the STeP tool. However, the exact pseudocode in those proofs differs from the pseudocode in [29] and [30].

6 Conclusions

We have presented process-algebraic models of safe, regular and atomic multi-writer multi-reader registers and used them to determine the robustness of various mutual exclusion algorithms for relaxed atomicity assumptions. Our analyses revealed issues with several of the algorithms discussed.

There are many more mutual exclusion algorithms that could be analysed in the same way as the ones shown in Section 5. In [30], Szymanski presents three other mutual exclusion algorithms. There also exist several variants of Szymanski's algorithm [23, 31], all of which are similar to the 3-bit linear wait algorithm but differ in small ways. In [6] it is shown that Dekker's algorithm does not ensure starvation freedom when safe registers are used and a modified version of the algorithm is presented which does satisfy this property. When we add verification of starvation freedom to our analysis, we can confirm their work.

We have only considered to what extent various algorithms guarantee mutual exclusion and whether the critical section is always reachable for every thread. Our next step will be to consider starvation freedom. Van Glabbeek proves that starvation freedom cannot hold for any mutual exclusion algorithm for which the correctness, on the one hand, relies on atomicity of memory interactions and, on the other hand, does not rely on assumptions regarding the relative speeds of threads [10]. A crucial presupposition for his argument is that a convincing verification hinges on not more than a component-based fairness assumption called justness. In [5] a method is proposed for verifying liveness properties under justness assumptions using the mCRL2 toolset. The method requires a classification of the roles of components in interactions. It should be investigated how to classify the roles of threads and registers in invocations and responses, and, in particular, how to deal with the ow_i, ew_i and er_i actions in the method.

References

1 K. Alagarsamy. Some myths about famous mutual exclusion algorithms. *ACM SIGACT News*, 34(3):94–103, 2003.

2 James H. Anderson and Mohamed G. Gouda. Atomic semantics of nonatomic programs. *Information Processing Letters*, 28(2):99–103, 1988.

3 Alex A. Aravind. Yet another simple solution for the concurrent programming control problem. *IEEE Transactions on Parallel and Distributed Systems*, 22(6):1056–1063, 2010.

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

5 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.

6 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.

7 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. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 21–39. Springer, 2019.

8 Franco Cicirelli and Libero Nigro. Modelling and verification of mutual exclusion algorithms. In *2016 IEEE/ACM 20th International Symposium on Distributed Simulation and Real Time Applications (DS-RT)*, pages 136–144. IEEE, 2016.

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

10 Rob van Glabbeek. Modelling mutual exclusion in a process algebra with time-outs. *CoRR*, abs/2106.12785, 2021. URL: https://arxiv.org/abs/2106.12785, arXiv:2106.12785.

11 Jan Friso Groote and Jeroen J. A. Keiren. Tutorial: designing distributed software in mCRL2. In *Formal Techniques for Distributed Objects, Components, and Systems: 41st IFIP WG 6.1 International Conference, FORTE 2021, Held as Part of the 16th International Federated Conference on Distributed Computing Techniques, DisCoTec 2021, Valletta, Malta, June 14–18, 2021, Proceedings*, pages 226–243. Springer, 2021.

12 Jan Friso Groote, Jeroen J. A. Keiren, Bas Luttik, Erik P. de Vink, and Tim A. C. Willemse. Modelling and analysing software in mCRL2. In Farhad Arbab and Sung-Shik Jongmans, editors, *Formal Aspects of Component Software - 16th International Conference, FACS 2019, Amsterdam, The Netherlands, October 23-25, 2019, Proceedings*, volume 12018 of *Lecture Notes in Computer Science*, pages 25–48. Springer, 2019. doi:10.1007/978-3-030-40914-2_2.

13 Jan Friso Groote and Mohammad Reza Mousavi. *Modeling and Analysis of Communicating Systems*. MIT Press Ltd., 2014.

14 Wim H. Hesselink. Mechanical verification of Lamport's bakery algorithm. *Science of Computer Programming*, 78(9):1622–1638, 2013.

15 Wim H. Hesselink. Mutual exclusion by four shared bits with not more than quadratic complexity. *Science of Computer Programming*, 102:57–75, 2015.

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

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

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

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

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

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

22 Leslie Lamport. The TLA+ Hyperbook, August 2015. Available at http://lamport.azurewebsites.net/tla/hyperbook.html, accessed on 26 April 2023, see Chapter 7.8.4.

23 Zohar Manna, Anuchit Anuchitanukul, Nikolaj Bjorner, Anca Browne, and Edward Chang. STeP: The Stanford Temporal Prover. Technical report, Stanford University Department of Computer Science, 1994.

24 Zohar Manna and Amir Pnueli. An exercise in the verification of multi-process programs. *Beauty is our business: a birthday salute to Edsger W. Dijkstra*, pages 289–301, 1990.

25 Radu Mateescu and Wendelin Serwe. A study of shared-memory mutual exclusion protocols using cadp. In *International Workshop on Formal Methods for Industrial Critical Systems*, pages 180–197. Springer, 2010.

26 Gary L. Peterson. Myths about the mutual exclusion problem. *Inf. Process. Lett.*, 12(3):115–116, 1981. doi:10.1016/0020-0190(81)90106-X.

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

28 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. arXiv:https://doi.org/10.1137/07071158X, doi:10.1137/07071158X.

29 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 1988, Saint Malo, France, July 4-8, 1988*, pages 621–626. ACM, 1988. doi:10.1145/55364.55425.

30 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 1990, Jerusalem, October 22-25, 1990*, pages 110–117. IEEE Computer Society, 1990. doi:10.1109/JCIT.1990.128275.

31 Boleslaw K. Szymanski and Jose M. Vidal. Automatic verification of a class of symmetric parallel programs. In *IFIP Congress (1)*, pages 571–576, 1994.

A Proof of Proposition 11

This section presents our proof that every schedule σ satisfying the write-order condition can be simulated by our model R_r.

The return value of a read operation $r \in ops(\sigma)$ is either the initial value of the register or the write value associate with some write operation $w \in ops(\sigma)$ such that $r \not<_\sigma w$. Let σ be a schedule that satisfies the write-order regular register condition; the *reads-from mapping* ρ for σ is the mapping

$$\rho : reads(\sigma) \to writes(\sigma)$$

that associates with every $r \in reads(\sigma)$ its direct predecessor in $\mathcal{S}_r$ (recall that we have included a special write operation w_{init} in $ops(\sigma)$ that precedes all other operations, so that every $r \in reads(\sigma)$ indeed has a direct predecessor in $\mathcal{S}_r$).

▶ **Proposition 13.** *Let σ be a schedule satisfying the write-order regular register condition and let ρ be the associated reads-from mapping. Then*

- *$\rho(r) \in rel\text{-}writes(r)$,*
- *there does not exist a write $w \in writes(\sigma)$ such that $\rho(r) <_\sigma w <_\sigma r$, and*
- *the write value of $\rho(r)$ equals the return value of r.*

Our goal is now to show that every schedule satisfying the write-order regular register condition can be transformed into a trace of our regular register model by appropriately inserting ow_i actions.

▶ **Lemma 14.** *Let $w \in writes(\sigma)$. If the set $\rho^{-1}(w) = \{r \in reads(\sigma) \mid \rho(r) = w\}$ is infinite, then it has an infinite subset $R \subseteq \rho^{-1}(w)$ such that for all $r \in R$ and for all $w' \in writes(\sigma)$ we have that $w' <_\sigma r$.*

Proof. Let $W = \{w' \in writes(\sigma) \mid w \not<_\sigma w'\}$. Since the invocations of all $w' \in W$ must appear in σ before the response of w, we have that W is finite. At most finitely many reads can have their invocations appear before the last occurrence of a response of a write in W and so for infinitely many reads $r \in \rho^{-1}(w)$ we have that $w' <_\sigma r$ for all $w' \in W$. Let R be the set of all those reads, i.e.,

$$R = \{r \in \rho^{-1}(w) \mid \forall w' \in W. w' <_\sigma r\} \ .$$

It remains to argue that there cannot exist $w' \in writes(\sigma)$ such that $w <_\sigma w'$. To this end, we derive a contradiction from the assumption that there does exist such $w' \in writes(\sigma)$. Since only finitely many reads can have their invocations before the occurrence of the response of w' in σ (for the prefix of σ including the response of w' is finite), it follows that there exists $r \in R$ such that $w' <_\sigma r$. But then we have that $\rho(r) <_\sigma w' < r$, which contradicts the statements in Proposition 13. ◀

▶ **Definition 15.** *We say that a read $r \in reads(\sigma)$ is* non-overlapping *if $\rho(r) <_\sigma r$. We denote the set of all non-overlapping reads in σ by $reads_{no}(\sigma)$.*

▶ **Lemma 16.** *Let σ be a schedule that satisfies the write-order condition. Then there exists an enumeration $\vec{o} = o_0, o_1, o_2, \ldots$ of $reads_{no}(\sigma) \cup writes(\sigma)$ satisfying the following properties:*

1. *$o_0 = w_{init}$;*
2. *if $o_i <_\sigma o_j$, then $i < j$ for all relevant i, j;*
3. *for every $r \in reads_{no}(\sigma)$ we have that $\rho(r)$ appears before r in $\vec{o}$ and between $\rho(r)$ and r there is no other write; and*

4. *for all reads $r, r' \in \mathit{reads}_{no}(\sigma)$, if r and r' are distinct, $\rho(r) = \rho(r')$ and r appears before r' in $\vec{o}$, then the invocation of r occurs before the invocation of r' in σ.*

Proof. We define $\vec{o}$ in three steps: first, we define an enumeration of

$$W = \bigcup_{r \in \mathit{reads}(\sigma)} \mathit{rel\text{-}writes}(\sigma, r) \ ;$$

then we extend this enumeration to an enumeration of $\mathit{writes}(\sigma)$; and finally we suitably insert the elements of $\mathit{reads}_{no}(\sigma)$ in this enumeration. After defining $\vec{o}$ we shall establish that it satisfies the required properties.

Define the relation $\mathcal{S}$ on W by

$$\mathcal{S} = \bigcup_{r \in \mathit{reads}(\sigma)} (\mathcal{S}_r \cap (W \times W)) \ .$$

To prove that $\mathcal{S}$ is irreflexive, it suffices to note that $\mathcal{S}_r$ is irreflexive for all $r \in \mathit{reads}(\sigma)$.

To prove that $\mathcal{S}$ is transitive, let $w_1, w_2, w_3 \in W$ and suppose that $w_1 \mathrel{\mathcal{S}} w_2$ and $w_2 \mathrel{\mathcal{S}} w_3$. From the definition of $\mathcal{S}$ and $w_1 \mathrel{\mathcal{S}} w_2$ it follows that $w_2 \neq w_{init}$; similarly, from $w_2 \mathrel{\mathcal{S}} w_3$ it follows that $w_3 \neq w_{init}$. Then there exist r and r' such that $w_1 \mathrel{\mathcal{S}_r} w_2$ and $w_2 \mathrel{\mathcal{S}_{r'}} w_3$. If $r = r'$, then $w_1 \mathrel{\mathcal{S}} w_3$ immediately follows by the transitivity of $\mathcal{S}_r = \mathcal{S}_{r'}$. Otherwise, either the response of r' occurs later in σ than the response of r, or vice versa. In the first case, we have that $\mathit{rel\text{-}writes}(\sigma, r) \subseteq \mathit{rel\text{-}writes}(\sigma, r')$ and therefore we get by the write-order regular register condition that $w_1 \mathrel{\mathcal{S}_{r'}} w_2$; since $\mathcal{S}_{r'}$ is transitive, it follows that $w_1 \mathrel{\mathcal{S}_{r'}} w_3$ and hence $w_1 \mathrel{\mathcal{S}} w_3$. In the second case, we have that $\mathit{rel\text{-}writes}(\sigma, r') \subseteq \mathit{rel\text{-}writes}(\sigma, r)$ and therefore we get by the write-order regular register condition that $w_2 \mathrel{\mathcal{S}_r} w_3$; since $\mathcal{S}_r$ is transitive, it follows that $w_1 \mathrel{\mathcal{S}_r} w_3$ and hence $w_1 \mathrel{\mathcal{S}} w_3$.

To see that for all $w, w' \in W$ we either have that $w \mathrel{\mathcal{S}} w'$ or $w' \mathrel{\mathcal{S}} w$, note that there exists $r \in \mathit{reads}(\sigma)$ such that w and w' are both relevant for r. Hence, either $w \mathrel{\mathcal{S}_r} w'$ or $w' \mathrel{\mathcal{S}_r} w$, so we have $w \mathrel{\mathcal{S}} w'$ or $w' \mathrel{\mathcal{S}} w$.

Thus, we have now established that $\mathcal{S}$ is a total order on W.

Let $W' = \mathit{writes}(\sigma) \setminus W$. If $W' \neq \emptyset$, then W must be finite, for the writes in W' are not relevant for any read in $\mathit{reads}(\sigma)$ and hence their invocations all occur after the responses of all reads in σ. This means that there are finitely many reads in σ, and that σ must have a finite prefix σ' in which all responses of all reads in σ occur. The writes in W must have their invocations before the occurrence of the response of some read in σ, and so all occurrences of invocations of writes in W must occur in the finite prefix σ' of σ. This means that W is finite.

Now, let $\vec{w} = w_0, w_1, w_2, \ldots$ be the enumeration of $\mathit{writes}(\sigma)$ that starts with an enumeration of W that is consistent with $\mathcal{S}$ (i.e., for all natural numbers i, j we have that $i < j$ implies $w_i \mathrel{\mathcal{S}} w_j$), and that, if W is finite, is followed by an enumeration of W' consistent with the order of the invocations of its elements in σ (i.e., if $w_i, w_j \in W'$, then we have that $i < j$ implies that the invocation of w_i occurs before the invocation of w_j in σ).

We proceed to extend $\vec{w}$ to an enumeration of all operations in $\mathit{reads}_{no}(\sigma) \cup \mathit{writes}(\sigma)$ by inserting directly after each w_i the elements of

$$\rho_{no}^{-1}(w_i) = \{r \in \mathit{reads}_{no}(\sigma) \mid \rho(r) = w_i\} \ .$$

Let $\vec{r_i} = r_{i,0}, r_{i,1}, r_{i,2}, \ldots$ be the enumeration of $\rho_{no}^{-1}(w_i)$ that is consistent with the order of the invocations of the reads in σ (i.e., for all relevant natural numbers j, k we have that $j < k$ implies that the invocation of $r_{i,j}$ precedes the invocation of $r_{i,k}$ in σ). Note that, by

Lemma 14, $\vec{r_i}$ can only be infinite if w_i is the last element of $\vec{w}$. So we can now define an enumeration $\vec{o}$ of the operations in $\mathit{reads}_{no}(\sigma) \cup \mathit{writes}(\sigma)$ as follows:

$$\vec{o} = o_0, o_1, o_2, \ldots = w_0, \vec{r_0}, w_1, \vec{r_1}, w_2, \vec{r_2}, \ldots \ .$$

We proceed to argue that $\vec{o}$ satisfies the required properties.

1. It is immediate by our assumptions about w_{init} that it is the least element with respect to $\mathcal{S}$. So it is the first element of the enumeration of W, and hence of $\vec{o}$.
2. To prove that $o_i <_\sigma o_j$ implies $i < j$ for all relevant i, j, we assume that $o_i <_\sigma o_j$ and distinguish seven cases.
   - If $o_i, o_j \in W$, then from $o_j \in W$ and $o_i <_\sigma o_j$ it follows that there exists $r \in \mathit{reads}(\sigma)$ such that $o_i, o_j \in \mathit{rel\text{-}writes}(\sigma, r)$. Then $o_i \ \mathcal{S}_r \ o_j$, so $o_i \ \mathcal{S} \ o_j$ and hence o_i appears before o_j in $\vec{o}$.
   - If $o_i, o_j \in W'$, then from $o_i <_\sigma o_j$ it follows that the response of o_i, and hence also the invocation of o_i, appears before the invocation of o_j, and therefore o_i appears before o_j in $\vec{o}$.
   - If $o_i \in W$ and $o_j \in W'$, then it is immediately clear from the definition of $\vec{o}$ that $i < j$.
   - We show that $o_i \in W'$ and $o_j \in W$ is impossible. For suppose it is, then, since $o_j \in W$, there exists $r \in \mathit{reads}(\sigma)$ such that o_j is relevant for r, while, since $o_i \in W'$ we have that $r <_\sigma o_i$. From $o_i <_\sigma o_j$ it follows that $r <_\sigma o_j$, contradicting that o_j is relevant for r.
   - Consider $o_i, o_j \in \mathit{reads}_{no}(\sigma)$. From $o_i <_\sigma o_j$ it follows that the response, and hence the invocation, of o_i appears before the invocation of o_j in σ, so if $\rho(o_i) = \rho(o_j)$, then it immediately follows that $i < j$. So we proceed with the assumption that $\rho(o_i) \neq \rho(o_j)$. Since o_i and o_j are non-overlapping reads, we have that $\rho(o_i) <_\sigma o_i$ and $\rho(o_j) <_\sigma o_j$, and from $o_i <_\sigma o_j$ it, moreover, follows that $\rho(o_i) <_\sigma o_j$. Since both $\rho(o_i)$ and $\rho(o_j)$ are relevant for o_j, we have that $\rho(o_i)$ and $\rho(o_j)$ are ordered by the o_j-serialisation $\mathcal{S}_{o_j}$. Moreover, we must have $\rho(o_i) \ \mathcal{S}_{r_j} \ o_j$ and $\rho(o_j) \ \mathcal{S}_{r_j} \ o_j$ and since $\rho(o_j)$ must be the direct $\mathcal{S}_{r_j}$-predecessor of o_j it follows that $\rho(o_i) \ \mathcal{S}_{r_j} \ \rho(o_j)$. Hence, $\rho(o_i)$ appears before $\rho(o_j)$ in $\vec{o}$ and therefore $i < j$.
   - If $o_i \in \mathit{writes}(\sigma)$ and $o_j \in \mathit{reads}_{no}(\sigma)$, then both o_i and $\rho(o_j)$ are relevant for o_j. Since $\mathcal{S}_{o_j}$ must be consistent with $<_\sigma$ we have that $o_i \ \mathcal{S}_{o_j} \ o_j$. Furthermore, since $\rho(o_j)$ is defined as o_j's direct predecessor according to $\mathcal{S}_{r_j}$, it follows that either $o_i \ \mathcal{S}_{o_j} \ \rho(o_j)$ or $o_i = \rho(o_j)$. In both cases, we find that $i < j$.
   - Suppose that $o_i \in \mathit{reads}_{no}(\sigma)$ and $o_j \in \mathit{writes}(\sigma)$. If $o_j \in W'$, then $\rho(o_i) <_\sigma o_i <_\sigma o_j$, so according to the definition of $\vec{o}$, o_i will appear in $\vec{o}$ between $\rho(o_i)$ and o_j, from which it follows that $i < j$. If $o_j \in W$, then there exists a read r such that $o_j \in \mathit{rel\text{-}writes}(\sigma, r)$. Since $\rho(o_i) <_\sigma o_i <_\sigma o_j$, it follows that $\rho(o_i)$ is also relevant for r. We must have $\rho(o_i) \ \mathcal{S}_r \ o_j$, and, according to the definition of $\vec{o}$, o_i will appear between $\rho(o_i)$ and o_j, so $i < j$.
3. Let $r \in \mathit{reads}_{no}(\sigma)$. Then $r \in \rho_{no}^{-1}(\rho(r))$, so r is an element of the sequence of writes directly following $\rho(r)$. It follows that $\rho(r)$ appears before r in $\vec{o}$ and between $\rho(r)$ and r there is no other write.
4. Let $r, r' \in \mathit{reads}_{no}(\sigma)$ be distinct, suppose that $\rho(r) = \rho(r')$ and r appears before r' in $\vec{o}$. Then we have $r, r' \in \rho_{no}^{-1}(\rho(r))$, so r and r' both appear in the sequence of reads that directly follows $\rho(r)$ in $\vec{o}$. The reads in that sequence are ordered in accordance with the order of their invocations. It follows that the invocation of r occurs before the invocation of r' in σ. ◀

The enumeration delivered by Lemma 16 allows us to define a procedure that transforms a schedule σ into a trace $\alpha \in \mathcal{T}_r$ as follows. Simultaneously iterate through the enumeration and the schedule. If the current operation in the enumeration is a read, then we move forward in the schedule until after the invocation of r and move to the next operation in the enumeration. If the current operation in the enumeration is a write, then there are two cases: if the invocation of the write has already occurred, then we insert the associated order action in the schedule, move past the order action, and move to the next operation; if the invocation of the write has not yet occurred, then we first move in the schedule until right after the invocation of the write, insert the associated order action, move past the order action, and move to the next operation. This establishes Proposition 11.

B Mutual Exclusion Counterexamples

In this appendix we give in-depth discussions and example traces for our more interesting results presented in Section 5. For all the models used as well as the exact pseudocode we modelled to arrive at the conclusions in Table 1, we point to the examples included with the mCRL2 distribution. These can be found at the same link as the register model itself.

In the counterexample discussions, we at times refer to the "entry protocol" and the "exit protocol" of an algorithm. The former is the part of the algorithm before the critical section is entered, the latter is the part after the critical section is released.

B.1 Properties

In mCRL2 properties must be encoded in the modal μ-calculus. We encoded the mutual exclusion property as

$$\forall_{i,j\in\mathbb{T}}.(i \neq j) \Rightarrow (\\ [\mathit{true}^\star]\neg(\langle \mathit{crit}(i)\rangle \mathit{true} \wedge \langle \mathit{crit}(j)\rangle \mathit{true}))$$

and the reachability of the critical section property as

$$\forall_{i\in\mathbb{T}}.\neg(\langle \mathit{true}^\star \cdot \mathit{noncrit}(i)\rangle \nu X.\langle \overline{\mathit{crit}(i)}\rangle X)$$

B.2 Peterson

In Section 5.1 we demonstrate a counterexample for Peterson's algorithm using safe registers. Here, we note that the same counterexample is also valid for our regular register model: r_2 can still return a 0 because it overlaps with w_4; and by placing the *order_write* of w_4 before the *order_write* of w_3 we can also have thread 0 read a 1 at r_4. However, this counterexample is not valid for the write-order definition from [28]. Since both writes to *turn* are relevant for both the reads from *turn*, the two threads must agree on their respective order. For thread 0 to read a 1 in *turn*, it must be the case that $w_4 < w_3$. And since w_3 ends before r_2 starts, r_2 cannot possibly read a 0. This is effectively the same situation as in Figure 4b. We cannot conclude from this that Peterson's algorithm is correct under write-order, only that this specific counterexample is not valid under those assumptions.

B.3 Attiya-Welch

The algorithm we refer to as the Attiya-Welch algorithm is presented in both [4] and [28] as being Peterson's algorithm from [26]. While the algorithm indeed has some similarities

to Peterson's it is not identical and in particular behaves different when using non-atomic registers. Hence why we refer to it as the Attiya-Welch algorithm, rather than a version of Peterson's.

As shown in the table, the Attiya-Welch algorithm does ensure mutual exclusion when non-atomic registers are used. Of interest is that while the original presentation of the algorithm from [4] also ensures reachability of the critical section when using non-atomic registers, the version of the algorithm presented in [28] does not. The two algorithms are shown in Algorithm 3 and Algorithm 4 respectively.

Algorithm 3 Attiya-Welch algorithm for two threads as presented in [4]. Once again i is a thread's own id, j the id of the other.

```
1:  flag[i] ← 0
2:  await flag[j] = 0 ∨ turn = j
3:  flag[i] ← 1
4:  if turn = i then
5:      if flag[j] = 1 then
6:          goto line 1
7:  else
8:      await flag[j] = 0
9:  critical section
10: turn ← i
11: flag[i] ← 0
```

Algorithm 4 Attiya-Welch algorithm for two threads as presented in [28] Once again i is a thread's own id, j the id of the other.

```
1:  repeat
2:      flag[i] ← 0
3:      await flag[j] = 0 ∨ turn = j
4:      flag[i] ← 1
5:  until turn = j ∨ flag[j] = 0
6:  if turn = j then
7:      await flag[j] = 0
8:  critical section
9:  turn ← i
10: flag[i] ← 0
```

When using regular registers and two threads, we get the following counterexample for reachability of the critical section on the variant presentation:

1. Thread 1 gains uncontested access to the critical section. Because thread 0 is not competing, thread 1 can reach line 8 without issue.
2. Thread 1 starts its exit protocol with starting $turn \leftarrow 1$. This operation does not have its order-action yet. At this point, a read of *turn* can read both a 0 and a 1: 0 is the initial value and no *order_write* actions have occurred yet; 1 can be read because of overlap.
3. Thread 0 starts the competition. Whenever it reads $flag[1]$ it sees a 1, but it can read whatever value it needs from *turn* to get through the entry protocol. It escapes the first await-loop by reading $turn = 1$; it escapes the repeat-until loop by again reading $turn = 1$; it avoids the second await-loop entirely by reading $turn = 0$ on line 6.

4. The order-action for thread 1's write takes place, but the write is not finished yet.
5. Thread 0 enters the critical section.
6. In the exit protocol, thread 0 writes $turn \leftarrow 0$. The order- and finish-actions take place immediately. At this point, the most recent order-action on *turn* has the value 0, and the write by thread 1 is still active. So once again, reads of *turn* can read either a 0 or a 1.
7. Thread 0 finishes the exit protocol with $flag[0] \leftarrow 0$. It re-enters the competition. Just like before, even though $flag[1] = 1$, thread 0 can get through most of the entry protocol by reading the required values for *turn*. It gets until right after line 5, having just escaped the repeat-until loop. Unlike the previous execution of the entry protocol by thread 0, it reads $turn = 1$ on line 6, as a result it starts awaiting $flag[1]$ to be 0.
8. Thread 1 stops writing to *turn*. At this point, any read of *turn* will give the value of the most recent order-action, which was 0.
9. Thread 1 end the exit protocol by setting $flag[1]$ to 0. Thread 0 does not read $flag[1]$ yet.
10. Thread 1 re-enters the competition. Since *turn* is now 0, it can get through most of the entry protocol. It cannot get through the entry protocol entirely however because on line 6, it once reads $turn = 0$. It then has to start awaiting $flag[0]$ to be equal to 0.
11. At this point, both threads have their respective *flag*s set to 1, and both are waiting for the other's *flag* to be 0. This is a deadlock; neither thread will ever be able to reach the critical section again.

This same counterexample (ignoring the ordering of writes) also works for the safe registers. This counterexample does not work for the write-order definition from [28] (it is once again similar to Figure 4b), but it does hold for the weak definition. This contradicts the claim in [28] that this algorithm ensures starvation freedom under weak – if one thread can never reach the critical section again after having done a *noncrit* action, then starvation freedom cannot hold.

As stated earlier, this counterexample is only present in the presentation from [28], it is not present in the pseudocode given in [4]. At first glance, the algorithms seem to be equivalent: the goto-statement on line 6 of Algorithm 3 is removed, and instead that part of the code is turned into a logically equivalent repeat-until loop. There is only a minor implementation difference: where in the original presentation *turn* is read only once to determine whether the loop should be taken and whether the thread needs to wait for the other to lower its flag; in the variant presentation, *turn* is read twice.

As shown in the counterexample, the deadlock requires an overlapping write on *turn* which can only occur when both threads are in the exit protocol simultaneously. Since mutual exclusion is guaranteed, it must be the case that one thread writing to *turn* is what allows the other thread to reach the exit protocol. In Algorithm 3 this is not possible: on line 4, if $turn = i$ is read then, since $flag[j] = 1$, the other thread will be forced back to line 1. If on the other hand $turn = j$ is read, then the thread gets stuck in the waiting loop on line 8. In Algorithm 4 it is possible: by reading $turn = j$ on line 5 and $turn = i$ on line 6. This is mentioned in part 3 in the counterexample.

B.4 Lamport (3-bit)

Lamport highlighted the (theoretical) importance of mutual exclusion algorithms that are resistant to safe registers [19]. He also proposed the first algorithm specifically designed to ensure mutual exclusion under safe registers: the Bakery algorithm [17]. But the Bakery algorithm requires unbounded memory registers and is therefore in many ways impractical and at the very least hard to analyse. In [18], Lamport proposes 4 different solutions for this

problem, each offering a different trade-off between using a small number of communication variables and satisfying stronger properties. We focus on the second algorithm, which requires only three SWMR bits per thread and still ensures both mutual exclusion and starvation freedom. The algorithm can be seen in Algorithm 5. We first introduce the variables used in the algorithm and the required definitions for understanding it.

This algorithm is for an arbitrary number of threads. We use id's 0 to $N-1$ when there are N threads. The j, y and f variables are private variables in the range 0 to $N-1$. The x_i, y_i and z_i registers are all Boolean variables initially set to 0/false.

Lamport's Three Bit Algorithm makes extensive use of cycles. A cycle, as defined in [18], is an object of the form $\langle i_0, .., i_m \rangle$ of distinct elements. Two cycles are the same if they contain the same elements in the same order except for a cyclic permutation. The first element of a cycle is its smallest element, so we take as the representative of a cycle a list where the smallest element is at index 0. An ordered cycle has all elements in order from smallest to largest, possibly only after cyclic permutation. Since our representation of a cycle is a list with the smallest element at the first position, an ordered cycle can be represented with a sorted list.

The operation ORD S takes a set S and returns the ordered cycle containing exactly the elements from S.

In the algorithm, the Boolean function $CG(v, \gamma, i_j)$ is used. Here, v is a Boolean function mapping each element in the cycle γ to either true or false, and i_j is an element from γ.

$$CG(v, \gamma, i_j) \stackrel{\text{def}}{\equiv} v(i_j) \equiv CGV(v, \gamma, i_j)$$

$$CGV(v, \gamma, i_j) \stackrel{\text{def}}{\equiv} \begin{cases} \neg v(i_{j-1}) & \text{if } j > 0 \\ v(i_m) & \text{if } j = 0 \end{cases}$$

The phrase "$i \leftarrow j$ **cyclically to** k" means that the iteration starts with $i = j$, then j gets incremented by 1, modulo the length of the cycle. This continues until $j = k$, at which point the iteration stops without executing the loop with $j = k$. $\oplus$ is used for addition modulo the length of the cycle.

Algorithm 5 Lamport's Three Bit algorithm

```
1:  y_i ← 1
2:  x_i ← 1
3:  γ ← ORD{i | y_i = 1}
4:  f ← minimum{j ∈ γ | CG(z, γ, j) = 1}
5:  for j ← f cyclically to i do
6:      if y_j = 1 then
7:          if x_i = 1 then x_i ← 0
8:          goto line 3
9:  if x_i = 0 then goto line 2
10: for j ← i ⊕ 1 cyclically to f do
11:     if x_j = 1 then goto line 3
12: critical section
13: z_i ← 1 − z_i
14: x_i ← 0
15: y_i ← 0
```

We find that the algorithm indeed satisfies mutual exclusion and reachability of the

critical section when using non-atomic registers. However, in the process of modelling it became clear these results are only valid when the computation on line 4 is implemented in a very particular manner, something which is not emphasised in the algorithm's presentation. In our first model, we handled this line as follows: we went through the constructed ordered list γ from the smallest to the largest element. For each, we read the associated z-value as well as the z-value of the element it has to be compared against according to the CGV function. As soon as we found one which satisfies the equality, that value was chosen for f. This seems to match the description of the algorithm from [18], "evaluating $[z]$ at j requires a read of the variable z_j." Yet, this implementation leads to a violation of the reachability of the critical section, even when atomic registers are used.

The violation is caused when one thread is updating its z-value in the exit protocol, while a different thread has to read this z-value multiple times for the computation on line 4. Consider the following situation with two threads and atomic registers:

1. Thread 1 executes the entry protocol successfully because it is the only competing thread. It gets to its exit protocol and sets z_1 to 1. It then completes the rest of the exit protocol.
2. Thread 1 starts the competition again. Once again, it is the only competing thread and hence can reach the critical section and start the exit protocol. It has not yet updated z_1 to be 0 again.
3. Thread 0 starts the competition. At line 3 it finds that $y_0 = y_1 = 1$ so $\gamma = [0, 1]$.
4. On line 4, thread 0 tests if $CG(z, \gamma, 0) = 1$. For this, it needs to compare z_0 and z_1. It finds $z_0 = 0$ and $z_1 = 1$. We find $CG(z, \gamma, 0) = 0$, so 0 is not deemed a valid value for f.
5. Thread 1 now updates z_1 to be 0.
6. Next, thread 0 checks $CG(z, \gamma, 1)$. It once again reads z_0 and z_1, but now it wants those to have different values. It reads $z_0 = 0$ and $z_1 = 0$. Hence, $CG(z, \gamma, 1) = 0$ so 1 is not deemed a valid value for f.

The algorithm does not account for there being no valid value for f, because this situation is not meant to occur. As a consequence, in our model thread 0 simply cannot take actions anymore once this situation is reached and will therefore never reach the critical section again. In [18] a lemma is proven which states that there is always some i for which $CG(v, \gamma, i) = 1$. But that proof does not account for a value changing while the comparisons are being done.

This situation can be avoided by reading all the z values once before starting line 4, and storing the result locally. Indeed, once we modelled the algorithm using this approach we found the results shown in Table 1. This once again highlights the importance of minor implementation details when dealing with non-atomic registers.

B.5 Szymanski (flag)

As stated in Section 5.2, Szymanski does not claim the flag algorithm is valid when using non-atomic registers. He instead claims that when the *flag* register is implemented as three bits as shown in Table 2, the algorithm is resistant to regular registers. Pseudocode is given in [29] that incorporates this change, as well as other changes to make the algorithm resistant to thread failure and restarts. However, some formatting issues in the pseudocode presentation means we are not certain exactly what was intended there. Instead of modelling that pseudocode, we therefore made the translation from the flag-algorithm to a three bit implementation ourselves and modelled that. The result is shown in Algorithm 6.

Interestingly enough, we find that mutual exclusion no longer holds even with atomic registers when this change is made. With two threads, mCRL2 reports that mutual exclusion holds with atomic registers (although not with safe or regular). For three threads, the following counterexample is found:

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

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

Algorithm 6 Szymanski's flag algorithm implemented with bits

```
1:  intent[i] ← 1
2:  await ∀j. intent[j] = 0 ∨ door_in[j] = 0
3:  door_in[i] ← 1
4:  if ∃j. intent[j] = 1 ∧ door_in[j] = 0 then
5:      intent[i] ← 0
6:      await ∃j. door_out[j] = 1
7:  if intent[i] = 0 then
8:      intent[i] ← 1
9:  door_out[i] ← 1
10: await ∀j < i. door_in[j] = 0
11: critical section
12: await ∀j > i. door_in[j] = 0 ∨ door_out[j] = 1
13: intent[i] ← 0
14: door_in[i] ← 0
15: door_out[i] ← 0
```

1. Thread 2 runs through the entire algorithm until line 14. At this point, $intent[2] = 0$, $door\_in[2] = 1$ and $door\_out[2] = 1$.
2. Threads 0 and 1 can both get past line 2, since $door\_in[0] = door\_in[1] = 0$ and $intent[2] = 0$.
3. Thread 1 continues further, at line 4 it sees $intent[0] = 1$ and $door\_in[0] = 0$ so it has to execute lines 5 and 6. It can immediately escape line 6 however, because $door\_out[2] = 1$.
4. Thread 1 continues through lines 7, 8 and 9. On line 10, it sees $door\_in[0] = 0$ so it can enter the critical section.
5. Thread 0 continues on line 3. There is no thread with *intent* set to true and *door_in* to false, so it directly gets to line 9 and 10. Since it has the lowest thread id, it can immediately enter the critical section.

This counterexample relies on the resetting of the variables in the exit protocol happening separately, rather than all at once. The properties can be made true if the order of the resets in the exit protocol is changed. If the order is $door\_out, door\_in, intent$ then both properties hold with three threads and atomic registers. This is not a desirable solution however. While we did not analyse starvation freedom formally, reconsidering the above counterexample makes it easy to observe that the following scenario is possible if *door_out* and *door_in* are reset before *intent* is:

1. Thread 2 runs through the entire algorithm until it gets to the exit protocol. Here it resets *door_out* and *door_in* but not *intent* yet.
2. Threads 0 and 1 both get past line 2 because $door\_in[0] = door\_in[1] = door\_in[2] = 0$. On line 4, both see $intent[2] = 1$ and $door\_in[2] = 0$, meaning they both go to lines 5 and 6.
3. If thread 2 never chooses to re-attempt access of the critical section, then $door\_out[2]$ will never become true again. And so threads 0 and 1 will never escape line 6.

So while reachability of the critical section holds, it relies on thread 2 always wanting to re-enter the competition. We foresee no such issues if the order of resets is *door_out*, *intent*, *door_in*. Although further formal verification is needed to confirm this reset order has no complications.

Note that regardless of the exit protocol's exact implementation, the algorithm still does not ensure mutual exclusion with safe or regular registers. For example, with two threads and safe registers, mCRL2 generates the following counterexample for mutual exclusion:

1. Both threads 0 and 1 get through lines 1 and 2.
2. Thread 0 starts writing 1 to $door\_in[0]$, but does not finish this write yet.
3. Thread 1 continues through line 3. On line 4, it reads $door\_in[0] = 1$ with overlap. It also sees $door\_in[1] = 1$ so can continue to line 7.
4. Thread 1 can simply continue through lines 7 and 9. On line 10, it reads $door\_in[0] = 0$ with overlap, and can therefore enter the critical section.
5. Thread 0 finishes the write on line 3. On line 4 it sees $door\_in[0] = door\_in[1] = 1$ so it continues on line 7. It does 7, 9 and then on line 10 it does not need to check any other thread's *door_in* values because it has the lowest thread id. Thread 0 can enter the critical section.

This counterexample is also valid for SWMR regular registers, since those allow two reads overlapping the same write to first read the new and then the old value.

B.6 Szymanski (3-bit linear wait)

An alternative version of Szymanski's algorithm is the 3-bit linear wait algorithm from [30]. The pseudocode is presented in Algorithm 7.

Algorithm 7 Szymanski's 3-bit linear wait algorithm

```
1:  a_i ← 1
2:  for j ← 0 to N−1 do await s_j = 0
3:  w_i ← 1
4:  a_i ← 0
5:  while s_i = 0 do
6:      j ← 0
7:      while j < N ∧ a_j = 0 do j ← j + 1
8:      if j = N then
9:          s_i ← 1
10:         j ← 0
11:         while j < N ∧ a_j = 0 do j ← j + 1
12:         if j < N then s_i ← 0
13:         else
14:             w_i ← 0
15:             for j ← 0 to N − 1 do await w_j = 0
16:     if j < N then
17:         j ← 0
18:         while j < N ∧ (w_j = 1 ∨ s_j = 0) do j ← j + 1
19:     if j ≠ i ∧ j < N then
20:         s_i ← 1
21:         w_i ← 0
22: for j ← 0 to i − 1 do await s_j = 0
23: critical section
24: s_i ← 0
```

Surprisingly, our verification shows this algorithm does not ensure mutual exclusion even when using atomic registers. The counterexample once again requires at least three threads. The first counterexample generated by mCRL2 relies on reading w_j and s_j separately on line 18. This was likely unintended behaviour, but as far as we can tell it is not excluded in the algorithm's description in [30]; not to mention that enforcing this would require a semaphore which returns us to the issue of assuming a lower-level solution to the mutual exclusion problem.

For completeness, we still model a variant where a semaphore is used to protect every write to a w- or s-register, as well as the two reads on line 18, but nothing else. This leads to the following counterexample with three threads and atomic registers:

1. Threads 0, 1 and 2 all execute lines 1 and 2. We now have that all a's are 1, all w's 0 and all s's 0.
2. Thread 1 continues the competition. It executes lines 3 and 4, on line 5 it sees $s_1 = 0$ so it goes into the while-loop. On line 7 it sees $a_0 = 1$ so it breaks out with $j = 0$ which means the condition on line 8 evaluates to false and the condition on line 16 evaluates to true.
3. Thread 0 does the same. On line 7, it also breaks early because $a_2 = 1$ hence it also ends up in the body of the if-statement on line 16.
4. Thread 0 reads $w_0 = 1, s_0 = 0$ on line 18; it continues with $j = 1$.
5. Thread 1 reads $w_0 = 1, s_0 = 0, w_1 = 1, s_1 = 0$ on line 18, so it continues with $j = 2$.
6. Thread 2 now continues. It executes lines 3 to 6. On line 7, it finds that all a-values are 0, so it continues on line 9. Here it sets s_2 to 1. On line 11 it once again sees that all a-values are 0, so it continues on line 14 where it sets w_2 to 0.
7. Thread 1 reads $w_2 = 0, s_2 = 1$, hence it breaks out of the loop on line 18 with $j = 2$. The condition on line 19 evaluates to true so thread 1 executes both $s_1 \leftarrow 1$ and $w_1 \leftarrow 0$.
8. Thread 1 goes back to line 5. Here it sees $s_1 = 1$, so it goes to line 22. It reads $s_0 = 0$ and can therefore enter the critical section.
9. Thread 0 now reads $w_1 = 0, s_1 = 1$, so it breaks out of the loop on line 18 with $j = 1$. The condition on line 19 evaluates to true so it executes $s_0 \leftarrow 1$ and $w_0 \leftarrow 0$.
10. Thread 0 goes back to line 5 where it sees $s_0 = 1$. It then goes to line 22 where it does not need to check the s-value of any thread since it has the lowest id. Thread 0 can enter the critical section.

The issue is that the third thread allows the other two to satisfy exists-conditions when this should not be happening. This same counterexample is of course valid with regular and safe registers with three threads. We did find that with only two threads, the semaphore does ensure the two properties even when safe registers are used.