Safe registers and Aravind's BLRU algorithm for mutual exclusion in mCRL2

Myrthe Spronck

Supervisor: Bas Luttik

Abstract

Many algorithms for ensuring mutual exclusion rely on atomic shared registers, meaning that when there are concurrent interactions with the register it behaves as if those interactions were sequential. Because this requirement is not always met, mutual exclusion algorithms have been designed that do not have this reliance. This report concerns the modeling of the weakest type of useful shared registers as defined by Lamport, safe registers, in the model checker mCRL2. Specifically, we model multi-writer multi-reader safe registers as described in [1] . We use the safe register model to verify that Aravind's bounded least recently used algorithm for mutual exclusion works correctly when used with safe registers. We also show how Peterson's algorithm for mutual exclusion fails to ensure mutual exclusion when safe registers are used instead of atomic ones.

1 Introduction

1.1 Background

The mutual exclusion problem, first outlined by Dijkstra [2], describes a scenario where multiple threads try to access the same shared resource simultaneously but only one of them is allowed access to it at any one time. We can abstract away from the shared resource using the term "critical section" to refer to the part of the code that handles accessing the shared resource. A solution to the mutual exclusion problem is an algorithm that makes sure that threads can never execute their critical sections simultaneously.

The primary property for a mutual exclusion algorithm is that mutual exclusion is not violated. However, mutual exclusion algorithms usually give more guarantees, for instance a bound on how often a thread might be passed over in the competition for access to the critical section before it is granted access.

There are many mutual exclusion algorithms, the most well-known being Peterson's algorithm [3]. However, as Lamport pointed out in [4], many algorithms for mutual exclusion, including Peterson's, are dependent on interactions with registers being executed atomically. That is, the concurrent execution of read and write actions on the registers are assumed to have the same behavior as executing those actions sequentially. In effect, this means that the registers are expected to behave as if access to them happens in mutual exclusion.

This assumption that a form of mutual exclusion is already implemented at a lower level is not always reasonable, so we should be aware of when algorithms depend on it. Lamport proposed separating registers into three types, dependent on how they handle concurrent actions: safe, regular, and atomic [5]. Atomic registers are the ones previously discussed, where concurrent actions have no adverse effects. Safe registers have the weakest assumptions that Lamport suggests might still have some use: they satisfy the property that any read action that is not concurrent with a write action returns the most recently written value. We will give a more in-depth definition

of the behavior of safe registers in Section 3, including the behavior of multi-writer registers, which Lamport did not consider. Regular registers fall somewhere in between atomic and safe registers, but these will not be considered in this report.

There exist mutual exclusion algorithms that can ensure mutual exclusion even when only assuming safe registers. Lamport himself suggests the Bakery Algorithm [6], which is proven to guarantee mutual exclusion with safe registers. A weakness of the Bakery algorithm is that it requires integer values that can grow unboundedly large, something that cannot be implemented on a physical computer. Additionally, unboundedly large values make model-checking difficult. Aravind's bounded least recently used (BLRU) algorithm [7] is therefore of interest: it is designed to ensure mutual exclusion using only safe registers with bounded domains.

1.2 This Report

Our aim is to model the behavior of safe registers in the model checker mCRL2 and use this model to verify two mutual exclusion algorithms. We use a model of Peterson's algorithm to show how an algorithm that assumes atomic registers fails to ensure mutual exclusion when used with safe registers. Additionally, we model and verify Aravind's algorithm as an example of an algorithm that only assumes safe registers. Aravind's was chosen instead of Lamport's Bakery algorithm because of the bounded register domains.

In the coming sections, we will first give a more complete explanation of the concepts we will be modeling. Section 2 goes in-depth on the algorithms we have already mentioned: Peterson's algorithm and Aravind's algorithm are particularly relevant to this report, but an explanation of Lamport's Bakery algorithm is also included since it serves as the basis for Aravind's algorithm. Section 3 contains a full definition of safe registers, the rules defined there will form the basis of our model.

From Section 4 onward we will be diving into modeling in mCRL2, starting with a brief introduction the the mCRL2 toolset and language. We also include an explanation of the approach to design we use for each of the models that are part of this report. Next, in Section 5 we cover the mCRL2 model of safe registers in detail and show how it corresponds to the definition given in Section 3. Section 6 shows how Peterson's fails to ensure mutual exclusion when used with safe registers, this section both includes the counterexample provided by mCRL2 and an explanation of why this occurs. Sections 7 and 8 are both about the model of Aravind's algorithm, covering the specifics of the model and the results of verification respectively. The modal μ-calculus formulas for all the properties will be given and explained.

In Section 9 we discuss the quality of the safe register model and possible improvements that could be made. We also mention other properties that might be verified for Aravind's BLRU algorithm and another algorithm that would be interesting to verify using the safe register model. Finally, Section 10 contains our conclusion.

A note on the language used in this report: we will use the term "thread" to refer to an individual thread of execution that executes an algorithm, rather than the commonly used term "process". Instead, the term "process" will be reserved for mCRL2 processes, which are used to define the behavior of both registers and threads.

2 Algorithms

In the introduction three algorithms of significance have been mentioned: Peterson's algorithm for mutual exclusion, Lamport's Bakery algorithm and Aravind's BLRU algorithm for safe, bounded registers. In this section, these algorithms will be presented in detail. The specific variants of the algorithms as presented here, including variable names, are taken from Michel Raynal's book on concurrent programming [1].

First, it is useful to understand the structure of a mutual exclusion algorithm. They often make use of the concepts of an entry protocol and an exit protocol: the entry protocol is executed when a thread wants access to the critical section, it handles the competition for which thread gets access first. The exit protocol is called by a thread when it is done accessing the critical section, it takes the actions that allow another thread gain access next. If the algorithm works correctly, no other thread can access the critical section until the one that most recently had access has completed its exit protocol.

Algorithm 1 shows the code structure for a thread that continuously accesses the critical section.

Algorithm 1: Structure for a thread continuously trying to access the critical section

```
1 while true do
2     Non-critical Section
3     Entry Protocol
4     Critical Section Access
5     Exit Protocol
```

We will use the concepts of an entry protocol and an exit protocol in the discussion of the three algorithms.

2.1 Peterson's Algorithm

Peterson's algorithm [3] is a rather elegant solution to the mutual exclusion problem. This algorithm is not designed for dealing with safe registers, but it does serve as a clear example of how a mutual exclusion algorithm works. It is specifically made for two competing threads, it requires significant alterations to also serve more than two threads. Here, we only discuss the two-thread version.

Peterson's algorithm uses $flag$ variables for both threads, which they use to indicate that they are competing for the critical section. It also uses a shared variable $turn$ to decide which thread gains access to the critical section when both are competing.

Algorithm 2: Peterson's mutual exclusion algorithm for 2 threads, i, j. This is the code for i.

```
1 operation entry_protocol(i):
2     flag[i] ← true
3     turn ← i
4     wait flag[j] = false ∨ turn ≠ i
5 operation exit_protocol(i):
6     flag[i] ← false
```

The algorithm can be seen in Algorithm 2. In the entry protocol, the thread raises its flag (sets $flag$ to $true$) to signal it is competing. It then sets the $turn$ to its own id. In the wait, a thread first checks if the other thread is competing at all (if its flag is down, it is not). If it is not competing then no waiting is needed and the thread can immediately enter its critical section. If the other thread is also competing, then the value stored in $turn$ will be the id of the thread that wrote to $turn$ last. That means that of the two competing threads, exactly one will see a value in $turn$ that is not its own id, and enter the critical section. All of this is assuming that these registers are atomic, so no incorrect values will be read from the $turn$ register.

In the exit protocol, the thread only needs to signal that it no longer desires access to the critical section by lowering its flag. At that point, if the other thread was waiting it will be allowed to continue on to the critical section. Even if the thread that just finished its exit protocol tries to access the critical section again before the waiting thread has a chance, the $turn$ variable ensures that the waiting thread gets access first.

2.2 Lamport's Bakery Algorithm

Understanding the Bakery algorithm is useful for understanding Aravind's algorithm. For this reason, we include a detailed explanation of the Bakery algorithm here. However, modeling and verification of this algorithm will not be discussed in the rest of this report.

The idea behind the Bakery algorithm [6] is that of ticket numbers in a bakery, though a more contemporary comparison might be the numbers used to signal your order is ready at McDonald's. When someone makes their order (in this case, wants access to the critical section), they get a ticket number that is higher than the ticket numbers of those that came before. Then, the person waits until their number is called. In a bakery or McDonald's a quick order might be handled before a more complicated one, but for the comparison with this algorithm every order takes equally long to process, so whenever someone's number is called everyone with a lower number has already been serviced. This is a first-in first-out (FIFO) ordering.

The full algorithm is Algorithm 3.

Algorithm 3: Lamport's Bakery algorithm for n threads, this is the code for thread i

```
1 operation entry_protocol(i):
2     flag[i] ← true
3     my_turn[i] ← max(my_turn[0], ..., my_turn[n − 1]) + 1
4     flag[i] ← false
5     for j ∈ {0, ..., n − 1}, j ≠ i do
6         wait flag[j] = false
7         wait my_turn[j] = 0 ∨ ⟨my_turn[i], i⟩ < ⟨my_turn[j], j⟩
8 operation exit_protocol(i):
9     my_turn[i] ← 0
```

The registers my_turn are used in the algorithm to store a thread's ticket number. When a thread computes its ticket number, it first checks the highest number that any thread currently has and then adds one to it, so that is now has the highest number (line 3). Everyone's number is initialized to 0, and they set it back to 0 when they exit the critical section (line 9).

The Bakery algorithm also uses flags, just like Peterson's and Aravind's. However, the $flag$ value of a register is not used to signal its interest in accessing the critical section. Instead, a my_turn value greater than 0 indicates an interest. Indeed, while Raynal calls these registers "flag", Lamport's original name for them, "choosing", was more accurate for what they represent. They are used to signal to other threads that the thread is currently computing its ticket number. This is needed because checking the numbers of every other thread to get the maximum value takes time, during which the other threads might be executing their own actions.

The reason for the two waits in a row, lines 6 and 7, is as follows: thread i wants to compare its my_turn value to that of another thread, thread j. However, it cannot do this while j is still actively modifying its my_turn value, since these are safe registers and having an overlapping read and write on the same register ($my\_turn[j]$) will cause problems. So it first waits on $flag[j]$ being $false$ (line 6), since that indicated that thread j is not currently modifying its my_turn value.

In line 7, the check happens to see if thread j is either not competing for the critical section at all ($my\_turn[j] = 0$) or if it has a ticket number greater than the waiting thread's ticket number. If either one of those things is true, thread i wins the competition from thread j and can continue until it has won against all other threads (the for-loop at line 5), after which it can enter the critical section. The comparison $\langle my\_turn[i], i\rangle < \langle my\_turn[j], j\rangle$ is a comparison on a lexicographic ordering, where in the case of two my_turn values being equal, the thread id breaks the tie. The reason this comparison is used is that this algorithm does not ensure that every thread gets a different my_turn value, so a tie-breaker is needed. This comparison results in the

FIFO property of this algorithm. Specifically this is FIFO on pairs [1], with the assumption that if two threads have the same *my_turn* value, they "came in" simultaneously so their ordering in relation to each other does not matter.

A disadvantage of this algorithm comes from line 3, where the new *my_turn* value is computed. This line allows the *my_turn* value to grow unboundedly large. While ticket numbers do get reset to 0, as long as a thread starts computing its *my_turn* value before all other threads have reset to 0, it will still get a larger value than was seen since the last time all *my_turn* values were 0. This can even happen with just two threads; take this trace as example:

Threads T_0 and T_1 start with $my\_turn[0] = my\_turn[1] = 0$. Thread T_0 raises its flag, computes $my\_turn[0] = 1$ and lowers its flag. It passes all waits (since T_1 is not competing yet) and enters the critical section. Thread T_1 then raises its flag and computes $my\_turn[1] = 2$. It then starts waiting on T_0. T_0 exits the critical section, and resets $my\_turn[0] = 0$. Then, it immediately raises its flag back up and computes $my\_turn[0] = 3$. T_1 enters the critical section, since its ticket number is lower than that of T_0, and T_0 starts waiting for T_1 to exit the critical section.

This loop of one thread recomputing its *my_turn* value before the other has a chance to reset its own to 0 can be repeated indefinitely. This means that the *my_turn* registers should be unboundedly large.

2.3 Aravind's BLRU Algorithm

Aravind's BLRU algorithm [7] is very similar to Lamport's Bakery algorithm. His aim was to design an algorithm that retains the benefits of the Bakery algorithm without requiring unbounded registers. He called this algorithm Bounded Least Recently Used (BLRU) in his paper, because the unbounded variant of the algorithm satisfies the fairness property that of the competing threads, the thread that held the critical section least recently has an advantage in getting getting access first. The bounded variant does not fully satisfy LRU-fairness anymore, but it does approximate the property.

In his paper, Aravind refers to three arrays storing values for the threads: c, *stage* and ts. However, we will stick to Raynal's names and use $flag$, *stage* and *date*. The $flag$ register for a thread is used to signal that the thread is competing for access to the critical section, just like it is in Peterson's. The domains of the registers are as follows: $flag \in \{true, false\}$, $stage \in \{0, 1\}$, $date \in \{0, ..., N\}$ where N is the defined maximum value. The initial value for all $flag$ registers is $false$, and the initial values for all *stage* registers is 0. For the *date* registers, thread i starts with date i.

Algorithm 4: Aravind's bounded LRU algorithm for n threads, this is the code for thread i.

```
1  operation entry_protocol(i):
2      flag[i] ← true
3      repeat
4          stage[i] ← 0
5          wait ∀_{j≠i} : (flag[j] = false ∨ date[i] < date[j])
6          stage[i] ← 1
7      until ∀_{j≠i} : stage[j] = 0
8  operation exit_protocol(i):
9      date[i] ← max(date[0], ..., date[n − 1]) + 1
10     if date[i] ≥ N then
11         ∀_{j∈[0...n−1]} : date[j] ← j
12     stage[i] ← 0
13     flag[i] ← false
```

The full algorithm is Algorithm 4. The reasoning is as follows: the algorithm requires threads to proceed through two logical stages to gain access to the critical section, and *stage* is used to

denote in which stage a thread currently is. When the *stage* value of a thread is 1, it has found that no other competing thread has a *date* value lower than it (checked in line 5), hence it believes it is its turn to enter the critical section. However, due to these registers being only safe, multiple threads could arrive at this conclusion. Therefore, a thread that is in stage 1 will check whether there are other threads that are also in stage 1 (checked in line 7). If these are indeed found, it repeats the repeat-loop: resets to stage 0 and checks the dates of the other competing threads again. Only when a thread is the only one in stage 1 does it gain access to the critical section.

Just like in Lamport's Bakery algorithm when a new *date* value is computed it is the maximum of all current dates plus 1 (compare line 9 in Aravind's algorithm to line 3 in the Bakery algorithm). In lines 12 and 13, it resets its *stage* and $flag$ to their default values. Resetting the *stage* allows other threads to pass the check on line 7, and resetting $flag$ allows other threads to continue if they were stuck waiting on this thread in line 5.

The variant of the algorithm presented here is the one where the *date* values are bounded. This bound is caused by lines 10 and 11, which reset the date of every thread to the initial value as soon as the date of the current thread becomes larger than or equal to some maximum value. This maximum value N should be picked so that the size of the domain of the *date* registers is greater than or equal to two times the number of threads. In our case, the domain runs from 0 to N, so we should pick $N \geq (2 \cdot n) - 1$ where n is the number of threads. Without lines 10 and 11, the algorithm still works but has unbounded *date*-registers instead, so $date \in \mathbb{N}$.

Remark 1. Note that in the unbounded version of the algorithm, the only thread that can write to $date[i]$ is thread i, meaning that these are single-writer registers. In the bounded version line 11 allows any thread to modify the date value of any other thread, so these are now multi-writer registers. The risks that come with multi-writer registers will be discussed in section 3, but know that this algorithm avoids the problems caused by concurrent writes to a single register by having these writes occur in the exit protocol. In order for a thread to be in the exit protocol, it must have just had access to the critical section. And since it has not completed its exit protocol, no other thread could have gained access to the critical section yet, so cannot have reached the exit protocol. As a result, the algorithm ensures that there can be no concurrent writes to the *date*-registers.

The resetting of the *date*-values is what causes the weakening of the LRU-property. In the unbounded algorithm, the thread that had access to the critical section the longest ago has the lowest *date*-value, since all threads that accessed the critical section after it have set their dates to a higher value. Since the entry protocol favors the competing thread with the lowest date, the LRU-property is satisfied. However, the resetting of dates means that we can no longer be sure that the thread with the lowest *date* had access to the critical section the least recent. However, Aravind proves in [7] that while a thread is waiting to gain access to the critical section, there can be at most one reset of the dates. As a result, a competing thread that had access to the critical section the least recent can lose the competition to gain access at most once to each other thread. This would be when the thread with the least recent access has the highest id, so when the reset happens it ends up with the highest *date*. As a result, while the LRU-property is no longer fully satisfied, it is still approximated.

3 Safe Registers

So far, we have only given a very brief introduction to the behavior of single-writer multi-reader (SWMR) safe registers. Here, we will define the exact behavior of multi-writer multi-reader (MWMR) safe registers, the kind that we will be modeling.

The only assumption that is made for safe registers is "a read not concurrent with any write yields the correct value—that is, the most recently written one" [5]. In his book on concurrent programming, Michel Raynal [1] expands this into a full description of the behavior of MWMR

safe registers. His definition is as follows:

1. A read that is not concurrent with a write operation (i.e., their executions do not overlap) returns the current value of the register.
2. A read that is concurrent with one (or several) write operation(s) (i.e., their executions do overlap) returns any value that the register can contain.
3. When write operations are not concurrent, an MWMR safe register behaves as SWMR safe register
4. When write operations are concurrent, the value written into the register is any value of its domain (not necessarily a value of one of the concurrent writes).

Lamport did not include definitions for MWMR registers in his paper, as a result these rules about concurrent write operations are not used by everyone. For instance, in [7], Aravind states that "no two writes on the same memory location overlap, but any other combinations may overlap", which seems to imply that he views concurrent writes as not being allowed by the system at all. However, since this assumption is stronger than the assumption that they can occur (with undesirable results) and safe registers should be the weakest variant, we will stick to Raynal's interpretation.

Both rules 2 and 4 mention that the arbitrarily returned values still belong to the domain of the register: "any value that the register can contain" and "any value of its domain" respectively. The assumption is made that even when incorrect values are returned, they are still in the register's domain. This is important to our model, since it means that even in case of concurrent operations there is still a bound on number of possible outcomes.

We need MWMR registers for the *date* registers in Aravind's algorithm and the *turn* register in Peterson's algorithm. Therefore we take rules 3 and 4 into account for the model.

4 Modeling Approach

The main focus of this article is modeling, so this section will introduce the model checker we use as well as some of the design decisions we made for all the models that are part of this report.

4.1 mCRL2

A model checker is a tool in which you define a formal model of a system and properties that you want to verify for that system. It establishes whether the property holds by exhaustively searching the entire state-space defined by the model for a counterexample to the property. If it cannot find one, then the property holds in every situation allowed by the model.

The model checker we use is mCRL2, a toolset developed at the department of Mathematics and Computer Science of the Technische Universiteit Eindhoven, in collaboration with the University of Twente. To define models, mCRL2 has its own language based on the Algebra of Communicating Processes, and modal μ-calculus is used to specify the properties [8].

Some mutual exclusion algorithms have already been modeled and analyzed in mCRL2. Peterson's mutual exclusion algorithm for two threads has already been modeled by Jan Friso Groote [9]. That model implements registers in the following way: by modeling them as a process, where the current state of the process stores what value is in the register, and the register processes can interact with the processes representing threads to model read and write actions. This model of registers treats them as atomic, but we built the model of safe registers as an extension of this.

There is also a model of Lamport's Bakery algorithm [10]. This model does not have registers as separate processes; rather it treats the values that should be stored in registers as parameters of the thread processes and has them communicate with each other to share information. As a

result of this, when thread 1 wants to read the flag value of thread 2, thread 2 needs to participate in this read. While this works to show how the algorithm is designed, it leaves the role of the registers and their capabilities rather implicit. Therefore we choose to not follow this style of register modeling.

4.2 Design Choices

For the sake of clarity, we set out a standard way of designing models. This also makes it easier to verify the same property on different models and to integrate the model of safe registers with models of different algorithms.

Most of this section is dedicated to stylistic choices like the naming of actions, and approach choices like what parameters the various processes get. However, an important point to make first is that there is one strong assumption we make in our models: we assume that threads cannot execute multiple interactions at the same time, i.e. if a thread starts an interaction its next action can only be ending that interaction. This restriction does not apply to the registers, they can handle multiple interactions at the same time. After all, we want to observe overlapping interactions on registers.

As already mentioned in Section 4.1, we stick to the approach of modeling registers as processes, where the parameters of the process includes at least a variable representing the current value written into the register. Most registers are associated with a thread (for instance, $flag[i]$ is associated with thread i), so for those registers an parameter id is also required. Processes representing threads also need the id of the thread as a parameter.

As an example of how we might model a thread using mCRL2 processes, we include the relevant processes from the model of Peterson's with safe registers (full model in Appendix B):

```
1  Thread(id: Nat) =
2      set_flag_start_t(id, id, true). set_flag_end_t(id, id)|wish(id).
3      set_turn_start_t(id, id). set_turn_end_t(id).
4      BusyWait(id);
5
6  BusyWait(id: Nat) =
7      get_flag_start_t(id, other(id)). sum flag_other: Bool.
8      get_flag_end_t(id, other(id), flag_other).
9      get_turn_start_t(id). sum turn: Nat. valid_id(turn) ->
10     get_turn_end_t(id, turn).
11     (!flag_other || turn != id) -> Critical(id) <> BusyWait(id);
12
13 Critical(id: Nat) =
14     enter(id). leave(id).
15     set_flag_start_t(id, id, false). set_flag_end_t(id, id).
16     Thread(id);
```

This is an example of how we use multiple processes to represent one thread. A *Thread* process is first started, but eventually it becomes a *BusyWait* process, which then becomes a *Critical* process. Each of these processes are defined by what sequence of actions can be taken when they are active.

In the example, most actions are of the shape $\{get, set\}\_\{flag, turn, date\}\_\{start, end\}\_t$. These actions are used for the interactions between threads and registers. The first part indicates the type of the interaction, whether it is a read (*get*) or a write (*set*). The second part is the type of the register being accessed. Suffixes *start* and *end* refer to whether this action represents the beginning or end of an interaction. This splitting of the start and end of interactions is not necessary when atomic registers are modeled, then interactions can be treated as occurring at a single time instant. But since we want to model safe registers, we need to be able to detect

overlapping executions. We allow time to pass between the start and end of an interaction, during which other threads can take their own actions.

The final part, $\_t$, refers to these actions being taken by a thread. This is part of how we model that interactions need to involve both a thread and a register. Registers have equivalent actions that end in $\_r$ instead, we can then define in mCRL2 that the thread and register halves of the interaction can only occur simultaneously. This allows us to model both sides of the interaction working together.

Data parameters are also added to the actions. These allow us to indicate which thread is accessing which register by adding both the thread id and the register id (if it is needed) to the actions. We can also use parameters to model the passing of data: at the beginning of a write action, the thread tells the register what value it wants to write, this is included as a parameter. Similarly, at the end of a read action the register returns a value to the thread. When these values are passed along, one part of the interaction does not know what value it will be. In the example, the thread knows what value it passes along to the register when it starts a write, but when it starts a read it does not know what value will be returned. To deal with this, we use the *sum*-operation.

Summations over variables are used to concisely define different options. For instance on line 9, the thread reads the *turn* value. The turn register contains either a 0 or a 1, we could write both options out explicitly, but instead we do a summation over a natural number *turn* (this is a variable name we picked, it could just as well have been n or x, it does not need to be the register name). mCRL2 automatically considers the different options and we can simply refer to the variable *turn* for the rest of the definition. There can sometimes be issues with the tool trying all values the defined domain, which in the case of natural numbers is infinite. Usually, mCRL2 will be able to figure out what values to consider by itself, but out of precaution we limit the domain ourselves by using calling a function that defines which values are in the domain. In this example, we use *valid_id* since the domain of the turn register is the id's of the threads. In other cases, we might use a function *in_domain* to define the size of a domain.

For read-interactions, the *sum* is placed after the start of the read. The reason for this is that if we placed it before the start of the read, then it would mean that the thread has already determined what value it will read before it even started. Not only is this inaccurate, it can also lead to deadlocks in the model if a read is started that can only end with a specific value, but that value is not available to be read at that point.

The actions related to the register interactions take up most of the model, but we also have three extra actions: *wish*, *enter* and *leave*, these represent a thread making its desire to access the critical section known, entering the critical section, and leaving the critical section respectively. We want to be able to distinguish which thread is doing these things, so we add the thread id as a parameter. Since *wish* is specifically for the register making it known that it wants access, we want it to occur simultaneously with the flag being raised in both Peterson's and Aravind's algorithms. We can indicate these two things happen simultaneously by making them a multi-action, this can be seen in line 2 of the example. It is part of the end of the interaction, because that is when the thread has finished making its desire known.

In this example we use the function *other* to get the id of the other thread. We will use this function again in the model of Aravind's algorithm. Functions like these make it possible to define the behavior of a general thread, rather than defining the two threads separately.

The only part of the model we have not covered yet is line 11: this line shows an if-then-else statement. The first part is a Boolean expression, in this case expressing that either the other thread is not competing or this thread was the first to set the *turn* value (this is the comparison on line 4 of Algorithm 2). If the expression evaluates to true, the part after "$-\,>$" is executed, in this case moving on to *Critical*. If it evaluates to false, the part after "<>" is executed, in this example the busy wait is repeated again.

5 Safe Register Model

With our overall approach defined, we can make the model of safe registers. Our aim is to represent the four rules given in Section 3 accurately.

Our model of a register is as follows:

```
1  Register(id: Nat, v:Nat, stat: Status) =
2      sum tid: Nat. valid_id(tid) -> (
3      sum n: Nat. in_domain(n) ->
4      (set_register_start_r(tid, id, n).
5      Register(stat=start_writer(stat, tid, n)))
6      +
7      get_register_start_r(tid, id).
8      Register(stat=start_reader(stat, tid))
9      +
10     (went_wrong(stat, tid) -> (
11         sum n': Nat. in_domain(n') -> (
12         set_register_end_r(tid, id).
13         Register(v=n', stat=end_writer(stat, tid))
14         +
15         get_register_end_r(tid, id, n').
16         Register(stat=end_reader(stat, tid))))
17     <> (
18         set_register_end_r(tid, id).
19         Register(v=stored_value(stat), stat=end_writer(stat, tid))
20         +
21         get_register_end_r(tid, id, v).
22         Register(stat=end_reader(stat, tid))))
23     );
```

This is an example of how we use a process to represent a register, in this case a register over natural numbers with the values bounded by the *in_domain* function. By altering the type of v, the *in_domain* function and the types mentioned in the Status object, the domain of the register can be altered. The full model incorporating this example, including an example thread that interacts with it and the exact function definitions, can be found in Appendix A.

As mentioned in Section 4.2, the register process has parameters for its id (id) and the currently stored value (v)[1]. This would be enough for modeling an atomic register, but we want to be able to keep track of some more information for a safe register, so we have a third parameter: the current status ($stat$). The status keeps track of three different things: the current number of active writers on the register; for each thread that is interacting with the register whether there was an overlapping write during its execution (in other words: if the interaction has gone wrong); and what value should be written into the register, assuming there is a write active and this write is executing correctly. Whenever the register starts or ends an interaction, the status object is updated using an update function. These functions are called *start_reader*, *start_writer*, *end_reader* and *end_writer*, their invocation can be seen on lines 5, 8, 13, 16, 19 and 22.

The exact definition of the status object and its associated update functions can be found in Appendix A. The only point we'll discuss in more detail here is how we can keep track of whether an interaction has gone wrong, since it might not be immediately obvious how to do this. According to the rules in Section 3, both read and write interactions "go wrong" when there is an overlapping write interaction. There are two scenarios: either at the time the interaction we want to assess starts there is already a write active, or a write starts during the execution of this interaction. (Or both, but there is no difference between an interaction that has one overlapping write and one that has several, so we do not need to consider that case on its own.) Both of these scenarios

[1] Or more precisely: the value written by the most recently ended write.

are easy to cover with our update functions: when a read or write interaction starts, the update function checks whether the current number of active writers is greater than 0. If it is, we know that there is an overlapping write so the interaction goes wrong. If the number of active writers is 0, we continue under the assumption that the interaction goes right. When a write interaction starts, we have it set the state of all other interactions to them having gone wrong. Since any interactions that were active at this time now encounter an overlapping write, we know they go wrong.

Returning to the model code, line 2 has a summation over all valid id's of threads, this means that we can define everything afterwards for an arbitrary thread *tid*, and mCRL2 will consider it defined for every thread that is part of our model.

The part of the code that actually defines the interactions between registers and threads is lines 3 to 22. Lines 3, 4 and 5 cover the the starting of a write: we need a *sum* to say that any value in the domain of the register could be written into it, then the *set_register_start_r* action can be taken, followed by the process updating its status and once again offering every interaction. Similarly, lines 7 and 8 cover the starting of a read interaction, though in this case no summation is needed since no values except id's are passed along at the start of a read.

Ending reads and writes is a bit more complicated, since we need to account for whether there was a concurrent write action during the execution. We resolve this as follows: on line 10 there is an if-then-else clause, which asks the status object whether the interaction belonging to this thread went wrong. If it did not, we continue with lines 18 to 22, which cover ending the write and updating the stored value v with the intended value (which was saved in the status object) or ending the read by returning the stored value v. These options correspond to rules 3 and 1 in Section 3 respectively. However, if the interaction did go wrong, we continue with lines 11 to 16. On line 11, there is another *sum*, since this following can happy for an arbitrary value in the domain. Then either the write ends and an arbitrary value is written into the register, in accordance with rule 4, or a read ends and returns an arbitrary value, in accordance with rule 2.

This safe register model represents a MWMR register, we include all 4 of the rules in Section 3. In the models for both Peterson's and Aravind's algorithms, we use it for all registers. Some register could be represented as SWMR registers instead, for instance the $flag$ registers in both algorithms, which are only written to by a single thread. We do not make a special model for this, because it does not make a difference whether we use the model of a MWMR register or a SWMR register in these cases. Because of our assumption that threads cannot carry out multiple interactions at the same time, a single thread cannot be responsible for two write interactions that overlap with each other. And since the algorithms only have a single thread writing to these registers, there will never be overlapping writes on them. As a result, we can use the MWMR model for SWMR registers without introducing behaviors that would not be there if we made a special SWMR model.

6 Peterson's Algorithm with Safe Registers

Before assessing Aravind's algorithm with safe registers, it is interesting to see an example of an algorithm that does work with atomic registers but not with safe ones. For this, we use Peterson's algorithm.

As mentioned in Section 4.1, a model of Peterson's for two threads in mCRL2 already exists. This model has atomic registers, and already shows that Peterson's indeed ensures mutual exclusion in this case [9]. We can make a variant of this model that includes safe registers. This requires a number of alterations: first and foremost replacing the register processes in the original model with the ones we developed in Section 5. Some slight alterations also have to be made in the model of the threads, to accommodate splitting read and write actions into start and end points. For consistency, we also replace the process and action names with the style defined in Section

4.2. The altered thread processes were included in that section already, and the full model for Peterson's for two threads with safe registers can be found in Appendix B.

Before we do the verification of the mutual exclusion property, we will need to define it in modal μ-calculus. To re-iterate: mutual exclusion is the property that at any time, at most one thread is in its critical section.When we translate that to the actions in our model we get: after a process executes the *enter* action, no other process can execute the *enter* action until the first has executed its *leave* action. Even more generally, we can say that once a process has executed its *enter* action, then until a process executes its *leave* action, no *enter* actions can be taken. In μ-calculus, this gives the property:

Property 1 (Mutual Exclusion)**.**

```
[true*.(exists id1: Nat. enter(id1)).
    (!exists id2 :Nat. leave(id2))*.
    (exists id3: Nat. enter(id3))]false
```

This definition of the mutual exclusion property was taken from [9].

With both the property and the model defined, we can ask mCRL2 to verify the property for us. As we might expect, the property no longer verifies now that we applied it with safe registers rather than atomic ones. mCRL2 can generate a counterexample for the property: an example of a sequence of actions (a trace) that the model can take that shows a violation of the property. This gives us the picture shown in Figure 1.

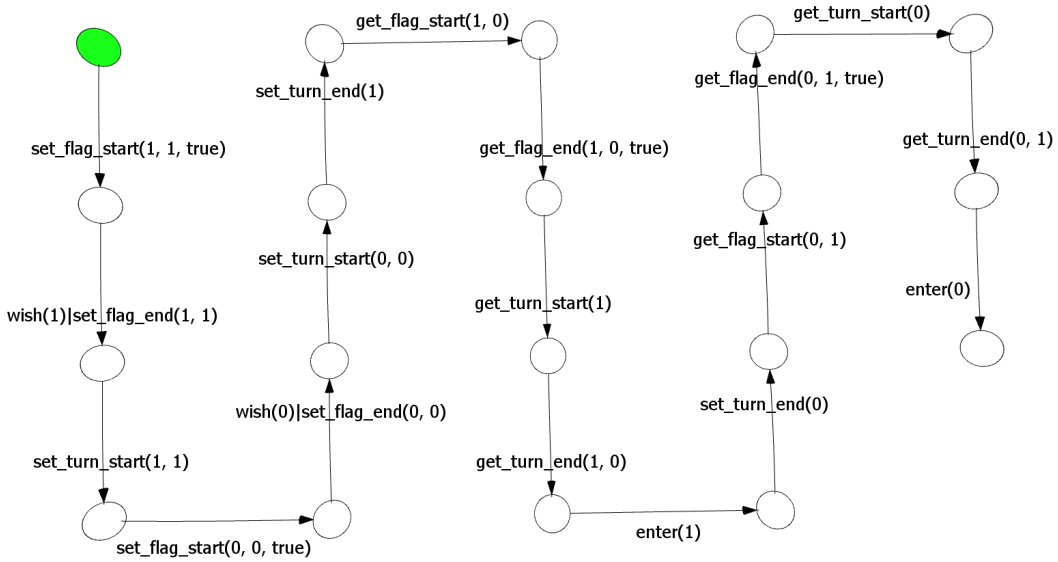

Figure 1: mCRL2 generated counterexample to mutual exclusion for Peterson's algorithm with safe registers

Interpreting this can be a bit difficult, but by looking at the start and end points of the interactions we can draw out how the interactions overlap. Figure 2 shows the result of this.

The problem is caused by the turn register: thread 0 starts writing to it while thread 1 is already doing so, leading to an arbitrary value being written into the register (by rule 4 in Section 3). Thread 1 stops writing to the register, and eventually starts reading it while thread 0 is still writing to it. This means that the read can return an arbitrary value (by rule 2), in this case 0. At this point, thread 1 has seen that the turn value value is not equal to its own id, so it enters the critical

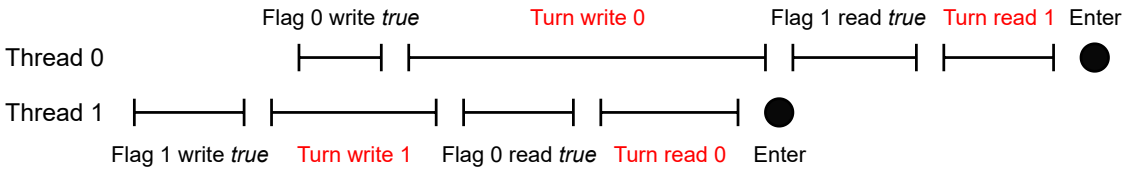

Figure 2: The trace from figure 1 drawn out for legibility, the interactions that cause the issue have been highlighted

section. Thread 0 then stops writing to the turn register, since there was an overlapping write an arbitrary value is written. In this case, we see later that the arbitrary value was a 1. Eventually thread 0 reads the turn register, this read goes correctly and returns the previously written value, 1. Thread 0 sees that the turn value is not equal to its own id, so it enters the critical section.

This trace is possible because turn is a MWMR safe register and Peteron's algorithm contains no safeguards to stop threads accessing it concurrently. We have shown that Peterson's algorithm no longer ensures mutual exclusion when we use it with safe registers.

7 Model of Aravind's Algorithm

This section is dedicated to the model of Aravind's BLRU algorithm for two threads. Here, we will discuss the part of the model that defines the processes that represent the actions threads take while executing the algorithm. The full model, including the register definitions and initialization, can be found in Appendix C.

For defining the thread processes, the only relevant thing to know about the registers are their domains. We have three different types of registers: *Stage*, *Flag* and *Date*. The *Flag* and *Stage* registers are both over Boolean values. This is a slight deviation from the algorithm as presented in Section 2.3, since there the stage registers are over the natural numbers 0 and 1. We simply use $false$ for 0 and $true$ for 1, this makes keeping track of the domains a bit easier. The *Date* registers are over a bounded set of natural numbers, we enforce this bound using the *in_domain* function, as established in Section 4.2. Since we have 2 threads, the domain of the *Date* registers is $\{0, 1, 2, 3\}$.

We use multiple processes to model a thread executing the algorithm. This is useful when dealing with the repeat-loop and busy wait in the algorithm. The loops require us to be able to repeat a previous part of the algorithm, which we do by referring back to the process representing that part. In total, the model includes 6 processes that represent parts of the algorithm: P, the moment the entry protocol is started; $R1, R2$ and $R3$ for the repeat-loop in the entry protocol; C for entering the critical section and starting the exit protocol; and CF for wrapping up the last bit of the exit protocol before going back to P. Each process only has a parameter for the id of the thread that it represents, they carry no other information.

We will discuss each of the 6 process definitions in turn, as well as specifically what lines of Algorithm 4 they correspond to. We first present them all together:

```
1  P(id: Nat) =
2      set_flag_start_p(id, id, true).
3      set_flag_end_p(id, id)|wish(id). R1(id);
4
5  R1(id: Nat) =
6      set_stage_start_p(id, id, false).
7      set_stage_end_p(id, id) . R2(id);
8
```

```
 9  R2(id: Nat) =
10      get_flag_start_p(id, other(id)).
11      sum flag_other: Bool. get_flag_end_p(id, other(id), flag_other).
12      get_date_start_p(id, id).
13      sum d: Nat. in_domain(d) -> get_date_end_p(id, id, d).
14      get_date_start_p(id, other(id)).
15      sum date_other: Nat. in_domain(date_other) ->
16      get_date_end_p(id, other(id), date_other).
17      ((flag_other == false || d < date_other) -> R3(id) <> R2(id));
18
19  R3(id: Nat) =
20      set_stage_start_p(id, id, true). set_stage_end_p(id, id).
21      get_stage_start_p(id, other(id)).
22      sum stage_other: Bool. get_stage_end_p(id, other(id), stage_other).
23      (stage_other -> R1(id) <> C(id));
24
25  C(id: Nat) =
26      enter(id). leave(id).
27      get_date_start_p(id, id).
28      sum d: Nat. in_domain(d) -> get_date_end_p(id, id, d).
29      get_date_start_p(id, other(id)).
30      sum date_other: Nat. in_domain(date_other) ->
31      get_date_end_p(id, other(id), date_other).
32      set_date_start_p(id, id, max(d, date_other) + 1).
33      set_date_end_p(id, id).
34      get_date_start_p(id, id).
35      sum d_res: Nat. in_domain(d_res) ->
36      get_date_end_p(id, id, d_res).
37      ((d_res >= LAST) -> (
38      set_date_start_p(id, id, id). set_date_end_p(id, id).
39      set_date_start_p(id, other(id), other(id)).
40      set_date_end_p(id, other(id)).
41      CF(id)) <> CF(id));
42
43  CF(id: Nat) =
44      set_stage_start_p(id, id, false). set_stage_end_p(id, id).
45      set_flag_start_p(id, id, false). set_flag_end_p(id, id) . P(id);
```

Firstly, process P is how a thread is initialized, at which point it immediately starts the entry protocol. Lines 2 and 3 of the model correspond to line 2 of Algorithm 4: setting $flag[id]$ to *true*. In the model, that is a write action, split into a start and an end, before we move on to $R1$. As mentioned in Section 4.2, the *wish*-action is taken simultaneously with the end point of raising the flag.

$R1$ corresponds to line 4 in the algorithm: setting the thread's own stage to 0, or in this case $false$. Once again, this is a simple write before moving on to $R2$. The reason this is in its own process is that when the when the condition to exit the repeat-loop is not met, the model needs to be able to return to this point.

$R2$ handles the wait part of the repeat-loop, line 5 of the algorithm. The first thing it does is read all the values it needs to be able to test whether it can stop waiting. First the process reads the $flag$ value of the other thread and stores this in the variable $flag\_other$ (lines 10 and 11 in the model). Then it reads its own *date* value and stores it in d (lines 12 and 13). Finally it reads the *date* value of the other thread and stores it in *date_other* (lines 14, 15 and 16). In line 17, the comparison is done to see if the conditions of the wait in the algorithm are met. If they are, we can move on to $R3$, else $R2$ needs to be repeated.

Remark 2. $R2$ represents a busy wait: if it does not read values that allow it to pass it tries to read the values again. This means that it is kept busy while not making progress. The introduction of a busy wait has an unfortunate side-effect: it allows the trace where one thread keeps infinitely repeating its busy wait while no other thread makes progress. While this is theoretically a scenario that could occur in real life, it is not realistic. In spite of this side-effect, we will keep this busy wait in since this is the most straightforward way to model the thread reading and comparing values. There are tricks in mCRL2 to only allow a process to read in values that would let it move on (if those values are not available, the process does not execute any actions until they are), but the way in which we split a read into a start and an end means that approach does not work here. A process already starts a read and therefore locks itself into waiting for that read to complete. Even if at that point some other condition occurred that would allow the process to move on, it still needs to complete that read first. Even if there is a way to remove the busy wait here, the repeat-loop also introduces infinite loops[2] so the problem persists regardless.

$R3$ is the first part of the model that handles more than one line in the algorithm. On line 20 of the model, it sets the thread's own *stage* to *true*, this corresponds to line 6 of the algorithm. Then, it checks whether the conditions are met to exit the repeat-loop. For this, it only needs to read the *stage* of the other thread, if it is *true* then the repeat-loop cannot be exited yet. This matches line 7 in the algorithm. If the loop can be exited, then the thread has gained access to the critical section, which we model by moving to C. Else, we return to $R1$ and the repeat-loop starts again.

C covers the thread entering and exiting the critical section, fittingly by taking the *enter* and *exit* actions (line 26). The rest of C handles the first parts of the exit protocol. First, it needs to read both *date* registers to find the maximum value, and then write that value plus 1 to $date[id]$. This is handled in lines 27 to 33 of the model. Then, it checks whether the value now written into the register (which it reads again) is greater than or equal to the defined maximum value, $LAST$, which in this case is 3. If it is not then it can immediately move on to CF, but if it is all the *date* values will need to be reset before it can move on. The comparison is handled on lines 34 to 37. Lines 38, 39 and 40 handle resetting the *date* values. Line 41 has both the move to CF after the *date*'s are reset, and the "else" part of the if-statement, which is just going to CF immediately.

Remark 3. There are a couple of reads and writes here that are superfluous. We could first calculate the result of the maximum of all the *date*-values, then before writing it into $date[id]$ check whether it exceeds the maximum. This would avoid the situation where we write a value and then immediately reset it. We could also remove the part reading back the value we just wrote and storing it in d_res. Instead, we could just do the comparison on $max(d, d\_other) + 1$, which would mean we could skip an entire read. However, our aim is to model the algorithm as set out by Aravind as closely as possible. In this case, that means making no assumptions about values being saved to private variables. It should be noted however that we did make the variant where the reading of d_res is skipped, and the resulting model is a bit smaller (about 4-6% fewer states) while still giving the same verification results. Unsurprising, since we know from Remark 1 that no other thread could be writing to $date[id]$ at this time, so the d_res is guaranteed to be equal to $max(d, d\_other) + 1$. Still, we want to keep the model as assumption-free as possible, so the extra read is kept.

Finally CF handles the last two lines from the exit protocol. This is setting the thread's own $stage$ and $flag$ to $false$. Then it returns to P to start the entry protocol again.

[2]This occurs if the competing thread with the lowest *date* keeps repeating because it keeps seeing a different thread is currently in *stage* 1. This could happen if that thread managed to get to *stage* 1 before the thread with the lowest *date* started competing and then never gets to set its own *stage* back to 0. Or if that other thread never actually got to *stage* 1, but has started the write on line 4 of the algorithm and never got to complete it. Rule 2 of safe register behavior means that the *stage* value could be read as 1.

8 Verification of Aravind's Algorithm

By combining the safe register model and the model of Aravind's BLRU algorithm (as shown in Appendix C), we have a complete model to verify properties on in mCRL2. In order to make defining properties easier, we use the hide-operation in mCRL2 to replace all actions except *wish*, *enter* and *leave* by τ (*tau*), the invisible action. The definitions of the eventual access and bounded overtaking properties were based on properties included with the model of Peterson's in the mCRL2 distribution [10].

Some properties require adding new actions to the model, the modified model used for those verifications is shown in Appendix E. Additionally, for properties that do not hold on the model, the mCRL2-generated counterexample can be found in Appendix F. In order to make these counter-examples legible, the actions are not hidden. Unless states otherwise, all verification is done on the model of two threads. A variant of the model that contains three threads exists as well (see Appendix D), but that model is too large to be suitable for doing much verification.

The first and most important property is **mutual exclusion**. We already defined this property in μ-calculus as Property 1 in Section 6. We use the exact same definition here. But where the property did not hold on Peterson's algorithm, it does hold on the model of Aravind's BLRU algorithm. This is one of the properties we also verified on the variant with three threads, mutual exclusion is satisfied there as well.

Secondly, we want to have **eventual access**, the property that when a thread expresses its desire to gain access to the critical section (represented by *wish*), it eventually gets it (represented by *enter*). In μ-calculus we can use the keyword *mu* to express that we want the smallest fixed point of states that satisfy some recursive formula. In this case, after a process has executed its *wish* action, we want the recursive formula to hold that either this thread can take the *enter* action in this state, or the current state is not a deadlock and this recursive property holds on all next states. This property only evaluates to true for a state where all possible paths from that state include this thread being capable of executing its *enter* action at some point. Our definition is as follows:

Property 2 (Eventual Access)**.**

```
[true*] forall id: Nat. [wish(id)] ( mu Y.
    (<enter(id)>true) || (<true>true && [true]Y) )
```

This property evaluates to false. The reason for this is addressed in Remark 2: the model of Aravind's algorithm contains both a busy wait and a repeat-loop. These lead to infinite loops in our state-space. If a process is infinitely trapped in such a loop, it will never be able to execute its *enter* action, and no other thread will be able to either, so this violates eventual access.

While eventual access in its pure form does not hold on the model, we can still verify whether it holds if we assume these infinite loops do not happen. This assumption is a fairness assumption, because the infinite loops we encounter occur in the event that one thread infinitely takes actions while no other thread gets to take any actions. It is "unfair" that while the other threads are capable of taking actions, they never actually get to on these paths. We define a modified eventual access property that ignores these infinite loops: **eventual access assuming fairness**. We can express this in μ-calculus by extending Property 2 so that instead of needing to eventually reach $\langle enter(id) \rangle true$, it needs to eventually reach that or an infinite loop. The most straightforward way to express an infinite loop on this model in μ-calculus would be to loop for an infinite sequence of τ actions, since we replaced every action except *wish*, *enter* and *leave* by τ. However, that would mean we take it for granted that the only infinite loops in the model are the ones we already know about. With just a slight modification of the model, we can be sure we only exclude infinite loops caused by the repeat and busy wait. To do this, we introduce a new action *Loop*, and put that as a multi-action with every action taken in $R1$, $R2$ or $R3$. Then we can look for an infinite sequence of *Loop* actions, rather than one of τ's:

Property 3 (Eventual Access Assuming Fairness)**.**

```
[true*] forall id: Nat. [wish(id)] ( mu X.
    ((nu Z. <Loop>Z) || <enter(id)>true) ||
    (<true>true && [true]X))
```

This property does hold for the model. So while eventual access is not guaranteed for every trace allowed by the model, if does hold if we ignore the unrealistic scenario where one thread infinitely loops without another thread taking any actions.

Those two properties cover the most important aspects of a mutual exclusion algorithm, but Aravind proves more properties for his algorithm in [7]. We verify those as well, to the extend that we can.

One property that Aravind proves his algorithm satisfied is **bounded overtaking**: the property that once a thread has made its desire to enter the critical section known, there is a bound on how many times another thread may gain access to the critical section before this thread does. Aravind claims that the bound for the BLRU algorithm is in the worst case $2n - 2$ where n is the number of threads. In the case of two threads, that means the bound on overtakes is 2. The property is defined by keeping track of how many times the other thread enters the critical section after a thread wishes for access. If this is more than twice in any trace, the property is violated. This is defined as follows:

Property 4 (Bounded Overtaking (bound = 2))**.**

```
[true*] forall id: Nat. [wish(id)]
    ( nu Y(n: Nat = 0). val(n<=2) &&
    [!enter(id)]Y(n) && [enter(other(id))]Y(n+1) )
```

This holds for our model, just as Aravind claims it should. Because the bound varies depending on the number of threads used, we also verify this property for the variant with three threads. Here we find that a bound of 2 is not satisfied, but a bound of 4 is. This fits, since $2 \cdot 3 - 2 = 4$.

A component of Aravind's proof of bounded overtaking is **bounded resets**, the property that while a thread is waiting for access to the critical section, it will experience at most one reset of its *date* value. While we have already verified bounded overtaking, this property is also interesting to verify. To do this, we add an extra action label *Reset*, which we put as a multi-action with the end of the write actions that are part of resetting a *date* value. We can then define a property very similar to the bounded overtaking property:

Property 5 (Bounded Resets (bound = 1))**.**

```
[[true*] forall id: Nat. [wish(id)]
    ( nu Y(n: Nat = 0). val(n<=1) &&
    [!enter(id)]Y(n) && [Reset(id)]Y(n+1) )
```

This evaluates to true. We can also use this property to test whether we really picked the smallest possible domain for our *date* registers. According to Aravind, the domain of the *date* registers should have size $\geq 2 \cdot n$ where n is the number of threads. We picked the domain $\{0, 1, 2, 3\}$ (see Section 7). But with this property defined, we can see what happens if we reduce the domain to just $\{0, 1, 2\}$. When we do this (by simply changing the value of $LAST$ from 3 to 2). both Property 4 and Property 5 evaluate to false. This tells us that having a domain smaller than $2 \cdot n$ indeed leads to the values being reset too often.

We also want to verify the LRU property, but as mentioned in Section 2.3 the bounded variant of this algorithm only satisfies a weaker version of LRU. Still, we can try to verify that the LRU property in its stronger form holds until the point a reset occurs. In his paper, Aravind states that the LRU property still holds for the bounded variant "when no timestamp [date] reset is encountered" [7], which together with Property 5 gives the approximation of LRU for the bounded variant.

In order to do a verification of LRU, we need to define exactly what it means for the algorithm to favor threads with lower *date* values. Here, we use the strictest definition of LRU: that when a thread gains access to the critical section, it has the lowest *date* value of all competing threads.

With this definition, we can verify **LRU until reset**. We again need to make some modifications to the model. For one, we need to be able to refer to whether a thread is competing in the μ-calculus formula. This information is stored in the $flag$ register, but we should have reliable access to the value, without the quirks of it being a safe register. To do this, we add an action $TrueFlag$ and extend the $Flag$ process to always be able to take this action with the corresponding id and value, we can then refer to which action is possible in the current state to get the real value of the $flag$ register. We also need to keep track of which thread has the true least recent access. Since we are verifying the model with two threads, this is much simpler than it would otherwise be: we add another process that stores which thread had the least recent access (initialized to an unused id, since at the start neither thread has had access at all) that updates its stored value whenever either thread takes the *enter* action by setting it to the id of the other thread. We use $TrueLeastRecentAccess$ similarly to $TrueFlag$ to get the id of the process that actually had the access the least recently. Finally, due to needing to update the register that stores the least recent access, we use $SeenEntering(id)$ instead of $enter(id)$. We can then say that from every state until we reach a reset, it should not be possible for a thread to enter the critical section if the other thread is competing and had access least recently:

Property 6 (LRU Until Reset)**.**

```
[true*] forall id: Nat. val(valid_id(id)) =>
    (nu X. <Reset(id)>true || ((
        [TrueLeastRecentAccess(other(id))]
        [TrueFlag(other(id), true)]
        [SeenEntering(id)]false
    ) && [true]X))
```

Contrary to what we would assume, this property does not hold. This is actually a quirk of the algorithm, not the model: say that thread 0 had access the least recent but is not competing for the critical section. Thread 1 starts the entry protocol and reaches line 5, where it sees thread 0 is not competing. Thread 0 then starts the entry protocol but does not get further than line 5, so it does not yet set its *stage* to 1. Then thread 1 can pass the check on line 7 and gain access to the critical section. LRU is then violated, because thread 0 is competing and has had access the least recent, but is passed over regardless. This trace does not rely on resets, it does not even rely on the registers being safe.

Aravind's proof that the unbounded version of the algorithm satisfies the LRU property does state that a thread with a lower *date* value that sets its $flag$ later "may" overtake the other thread [7], not that it is guaranteed to. Indeed, one wonders if it is even possible to make an algorithm that satisfies such a strict definition of the LRU property, since no matter how many checks the algorithm includes it is always possible that a thread that had the critical section less recently starts competing just a moment before the thread that won the competition enters the critical section.

Still, we have shown that Aravind's algorithm does not satisfy the strictest definition of LRU. The algorithm does give preference to threads with lower *date* values in the entry protocol, but we have shown that this preference is not strong enough to always let the thread with least recent access win the competition. A more nuanced definition of the LRU property will be needed to do further verification.

9 Discussion and Future Work

9.1 Quality of the Safe Register Model

In this section, we look at the quality of the safe register model and possible improvements that could be made. The primary concern is that the model accurately reflects the behavior of safe registers. We are confident that it does, the model accurately keeps track of overlapping write-interactions and we have demonstrated in Section 5 how it uses this information to follow the rules set out in Section 3.

However, there is a major concern with this model, which is the extreme state-space explosion upon applying it to more than two threads or to registers with large domains. Take for instance the model shown in Appendix A, which includes the safe register model and threads that read and write arbitrary values. When we look at the number of states and transitions in the state-space upon changing the number of threads and changing the register domain, we get Table 1.

		nr. of threads = nr. of registers			
		1	**2**	**3**	**4**
register	**1**	3 + 4	57 + 168	1 405 + 6 435	45 345 + 283 216
domain	**2**	8 + 12	332 + 1 328	18 488 + 120 024	1 367 312 + 12 303 616
size	**3**	15 + 24	1 017 + 4 896	94 635 + 758 889	11 819 601 + 132 667 632
	4	24 + 40	2 352 + 12 864	321 472 + 2 983 104	-
	5	35 + 60	4 625 + 27 800	862 625 + 8 922 375	-

Table 1: The number of states + transitions of the labeled transition system generated by the example model for safe registers. The last two fields became too large to generate.

Of course, these exact numbers depend on the behavior of the threads in this model, as well as how many registers we include per thread (in this case, one). However, this example still shows that this model grows much too fast to be suitable for verifying models with many threads and/or large register domains. As mentioned in Section 8, we could not do much verification on the model of Aravind's algorithm with three threads, because it grew too large. It went from needing 5 836 states and 13 240 transitions for two threads to needing 53 101 179 states and 207 813 391 transitions for three. This increase is due to the new thread that is initialized, the three registers that belong to that thread, and the increase in the *date* registers domain.

There are multiple factors to this problem. We suspect that part of it is the great number of possible ways the executions of different threads can interleave. In Remark 3, we pointed out that removing one read caused a decent reduction in the state-space of our model. Taking steps to reduce the number of register interactions in the model could therefore help with reducing the state-space. Another possibility, likely with much greater impact, would be to only treat write-interactions as taking up time and let read-interactions occur instantaneously. After all, as Aravind states in [11], "[the read of a safe shared variable] can be treated as atomic because it does not influence the shared state"; a read being active does not affect the result of overlapping writes or other reads. Therefore, instead of letting reads take time, we could check if at the moment of the read, a write is also active to decide the behavior of the read and then immediately end it.

Another source of the problem is the parameters a register needs to keep track of. States in the model are partially defined by the values in the parameters of the registers, so if there are unnecessary variables being stored this causes an increase in states without altering the accuracy of the model. During the development of the safe register model presented in this report, there was at some point a version that was the exact same, except that instead of storing one value for the intended write (if successful), it had room to store values for every thread. Of course, if more than one thread is trying to write simultaneously it does not matter what value was stored for them, an arbitrary value will be written either way, so this was unnecessary. When the switch was made to only have one stored value, there was a big improvement in the number of states

the model needed. With the safe register example model for two threads and two registers with a domain of size 2, we went from needing 1 421 states and 6 828 transitions to only needing 332 states and 1 328 transitions. We believe that the current number of variables is the minimum needed for this model, but it is possible that with a different model fewer parameters are needed.

When looking at more efficient models, we must be careful to not make the model less accurate. Early on in the development of this model, we designed a version that did not allow read or write interactions to take time. Rather, it let every interaction happen at just one time instant, and then used the mCRL2 mutli-actions to model the unreliable behavior of safe registers when reads or writes occur simultaneously with a write. However, under that model Peterson's algorithm satisfied mutual exclusion. The trace that leads to violation of mutual exclusion, as presented in Section 6, relies on one thread having multiple interactions with a register that go wrong because they overlap with just one write by another thread. If all interactions occur at just one time instant, this scenario is not possible. While this model did not suffer from the state-space explosion nearly as much as the current model, it was inaccurate to what we are modeling.

9.2 Further Verification

There are still properties of Aravind's algorithm that could be interesting to verify. In our analysis, we ran into problems verifying both the LRU property and the eventual access property. More work can be done to find out under what definitions of LRU Aravind's algorithm does meet the requirements, and how the bounded and unbounded variants differ in that regard. This will require giving very precise definitions for at what point exactly a thread starts competing, when we consider a thread as having won the competition, and what it means for one thread to overtake another.

Regarding eventual access, we had to make a fairness assumption to let the model satisfy the property. However, fairness assumptions are quite strong and not always warranted by reality. Justness assumptions have been proposed as an alternative to allow the verification of liveness properties without making such strong assumptions about the behavior of a system [12]. In [13], justness is defined in mCRL2 and used to show that Peterson's algorithm (with atomic registers) assures eventual access under justness assumptions. It would be interesting to see if Aravind also has this property, and if using atomic or safe registers affects the result.

Additionally, if a model can be made that suffers less from the state-space explosion, more verification could be done of Aravind's BLRU algorithm for more than two threads. This would give greater confidence that the algorithm satisfies the properties in general, not merely for two competing threads.

There is another algorithm that would be interesting to model with safe registers: in [14], Kessels presents a variant of Peterson's algorithm that only uses SWMR registers. That paper mentions that Peterson's algorithm requires "arbitrartion at a lower level" (meaning: the registers are atomic), which Kessels' algorithm does not need. This variant also uses bounded registers. The paper already contains a written verification that the algorithm works, but it would be interesting to verify it in mCRL2 as well.

10 Conclusion

The main contribution of this report is presenting the model of MWMR safe registers in mCRL2. We have shown that the model presented in Section 5 matches the definition of MWMR safe registers as given in [1]. We used the safe register model to examine the behavior of two different mutual exclusion algorithms when applied in a context where only safe registers are assured. We modified an existing model of Peterson's algorithm and showed that when using safe registers, it no longer ensures mutual exclusion. We also made a model of Aravind's BLRU algorithm and verified that it ensures mutual exclusion even when used with safe registers.

Additionally, we verified other properties of Aravind's algorithm. We showed that it satisfies eventual access under fairness assumptions, and confirmed Aravind's claim that the number of times a thread can be overtaken is at most $2n - 2$ where n is the number of threads. We also confirmed that a thread waiting for access to the critical section will encounter at most one reset of its *date* value. We also showed that the algorithm does not satisfy the strictest definition of LRU access, even before a reset of the *date* values occurs.

The model of a safe register presented here is rather general, its design is not dependent on the specific qualities of the algorithms discussed in this report. As a result, it can easily be used in models of other algorithms to verify their behavior under safe registers. While the model can lead to a large state-space explosion, we have suggested ways to counteract this that could be applied in future verifications.

References

[1] M. Raynal, *Concurrent Programming: Algorithms, Principles, and Foundations*. Springer Publishing Company, Incorporated, 2012.

[2] E. W. Dijkstra, "Solution of a problem in concurrent programming control," *Commun. ACM*, vol. 8, p. 569, Sept. 1965.

[3] G. Peterson, "Myths about the mutual exclusion problem," *Information Processing Letters*, vol. 12, pp. 115–116, 1981.

[4] L. Lamport, "The mutual exclusion problem: Part i—a theory of interprocess communication," *J. ACM*, vol. 33, p. 313–326, Apr. 1986.

[5] L. Lamport, "On interprocess communication-part i: Basic formalism, part ii: Algorithms," *Distributed Computing. Also appeared as SRC Research Report 8.*, pp. 77–101, December 1985.

[6] L. Lamport, "A new solution of Dijkstra's concurrent programming problem," *Commun. ACM*, vol. 17, p. 453–455, Aug. 1974.

[7] A. A. Aravind, "Yet another simple solution for the concurrent programming control problem," *IEEE Transactions on Parallel and Distributed Systems*, vol. 22, no. 6, pp. 1056–1063, 2011.

[8] J. F. Groote and M. R. Mousavi, *Modeling and Analysis of Communicating Systems*. MIT Press Ltd., 2014.

[9] J. F. Groote, J. J. A. Keiren, B. Luttik, E. P. de Vink, and T. A. C. Willemse, "Modelling and analysing software in mcrl2," in *Formal Aspects of Component Software* (F. Arbab and S.-S. Jongmans, eds.), pp. 25–48, Springer International Publishing, 2020.

[10] Technische Universiteit Eindhoven and University of Twente, "mCRL2 Distribution." `https://www.mcrl2.org/web/user_manual/index.html`. Accessed: 2021-02-01.

[11] A. Aravind and W. Hesselink, "Nonatomic dual bakery algorithm with bounded tokens," *Acta informatica*, vol. 48, pp. 67–96, Apr. 2011.

[12] R. V. Glabbeek and P. Höfner, "Progress, justness, and fairness," *ACM Computing Surveys*, vol. 52, Aug. 2019.

[13] M. Bouwman, B. Luttik, and T. Willemse, "Off-the-shelf automated analysis of liveness properties for just paths," *Acta Informatica*, vol. 57, pp. 551–590, Oct 2020.

[14] J. L. W. Kessels, "Arbitration without common modifiable variables," *Acta Informatica*, vol. 17, pp. 135–141, Jun 1982.

A Safe Register Model

```
% Defining the boolean array
sort
    ArrayB = Nat -> Bool;

map
    all_true: ArrayB;
    set: ArrayB # Nat # Bool -> ArrayB;
    get: ArrayB # Nat -> Bool;

var
    a: ArrayB;
    n: Nat;
    v: Bool;

eqn
    all_true(n) = true;
    set(a, n, v) = a[n -> v];
    get(a, n) = a(n);

% The status object
sort
    Status = struct status(ArrayB, Int, Nat);

map
    default_status: Status;
    start_reader: Status # Nat -> Status;
    start_writer: Status # Nat # Nat -> Status;
    end_reader: Status # Nat -> Status;
    end_writer: Status # Nat -> Status;
    stored_value: Status -> Nat;
    num_active_writers: Status -> Int;
    went_wrong: Status # Nat -> Bool;

var
    b: ArrayB;
    w: Int;
        s, v, id: Nat;

eqn
    default_status = status(all_true, 0, 0);
    start_reader(status(b, w, s), id) = status(set(b, id, w > 0), w, s);
    w > 0 -> start_writer(status(b, w, s), id, v) =
        status(all_true, w + 1, 0);
    w <= 0 -> start_writer(status(b, w, s), id, v) =
        status(set(all_true, id, false), w + 1, v);
    end_reader(status(b, w, s), id) = status(set(b, id, true), w, s);
    end_writer(status(b, w, s), id) = status(set(b, id, true), w - 1, 0);
    stored_value(status(b, w, s)) = s;
    num_active_writers(status(b, w, s)) = w;
    went_wrong(status(b, w, s), id) = get(b, id);

% Functions for limiting domain and id's
% limit set for id's should be the number of threads (+ associated register) you are using
% limit on domain is arbitrary in this example
map
    in_domain: Nat -> Bool;
    valid_id: Nat -> Bool;
var
    n: Nat;
eqn
    in_domain(n) = n < 3;
    valid_id(n) = n < 2;

% Action's used
act
    set_register_start_r, set_register_start_t, set_register_start: Nat # Nat # Nat;
    set_register_end_r, set_register_end_t, set_register_end: Nat # Nat;
    get_register_start_r, get_register_start_t, get_register_start: Nat # Nat;
    get_register_end_r, get_register_end_t, get_register_end: Nat # Nat # Nat;

% Processes
proc
    % Register process
    Register(id: Nat, v:Nat, stat: Status) =
        sum tid: Nat. valid_id(tid) -> (
        sum n: Nat. in_domain(n) -> (
        set_register_start_r(tid, id, n).  Register(stat=start_writer(stat, tid, n)))
        +
```

```
87          get_register_start_r(tid, id).  Register(stat=start_reader(stat, tid))
88          +
89          (went_wrong(stat, tid) -> (
90              sum n': Nat. in_domain(n') -> (
91              set_register_end_r(tid, id). Register(v=n', stat=end_writer(stat, tid))
92              +
93              get_register_end_r(tid, id, n'). Register(stat=end_reader(stat, tid))))
94          <> (
95              set_register_end_r(tid, id).
96              Register(v=stored_value(stat), stat=end_writer(stat, tid))
97              +
98              get_register_end_r(tid, id, v). Register(stat=end_reader(stat, tid))))
99      );
100
101     % Thread process, just an example for having interactions
102     Thread(id: Nat) =
103         sum rid: Nat. valid_id(rid) -> sum n: Nat. in_domain(n) ->
104         set_register_start_t(id, rid, n) . set_register_end_t(id, rid). Thread() +
105         sum rid: Nat. valid_id(rid) -> get_register_start_t(id, rid) .
106         sum n: Nat. in_domain(n) -> get_register_end_t(id, rid, n). Thread();
107
108 % Initialization, handles communication
109 % In this case, we have 2 threads that both have a register
110 init
111     allow({
112         get_register_start, get_register_end, set_register_start, set_register_end},
113     comm({
114         set_register_start_t|set_register_start_r -> set_register_start,
115         set_register_end_t|set_register_end_r -> set_register_end,
116         get_register_start_t|get_register_start_r -> get_register_start,
117         get_register_end_t|get_register_end_r -> get_register_end},
118     Thread(0) || Thread(1) ||
119     Register(0, 0, default_status) || Register(1, 0, default_status)
120     ));
```

B Model of Peterson's with Safe Registers

```
9  % Jan Friso Groote's model of Peterson's for two threads
10 % modified by Myrthe Spronck to include safe registers
11
12 % Arrays over boolean values
13 sort
14     ArrayB = Nat -> Bool;
15
16 map
17     all_true: ArrayB;
18     set: ArrayB # Nat # Bool -> ArrayB;
19     get: ArrayB # Nat -> Bool;
20
21 var
22     a: ArrayB;
23     n: Nat;
24     v: Bool;
25
26 eqn
27     all_true(n) = true;
28     set(a, n, v) = a[n -> v];
29     get(a, n) = a(n);
30
31 % two variants of the status object, depending on the type of the stored value
32 sort
33     StatusB = struct statusB(ArrayB, Int, Bool);
34     StatusN = struct statusN(ArrayB, Int, Nat);
35
36 map
37     default_statusB: StatusB;
38     start_reader: StatusB # Nat -> StatusB;
39     start_writer: StatusB # Nat # Bool -> StatusB;
40     end_reader: StatusB # Nat -> StatusB;
41     end_writer: StatusB # Nat -> StatusB;
42     stored_value: StatusB -> Bool;
43     num_active_writers: StatusB -> Int;
44     went_wrong: StatusB # Nat -> Bool;
45
46     default_statusN: StatusN;
47     start_reader: StatusN # Nat -> StatusN;
48     start_writer: StatusN # Nat # Nat -> StatusN;
```

```
49      end_reader: StatusN # Nat -> StatusN;
50      end_writer: StatusN # Nat -> StatusN;
51      stored_value: StatusN-> Nat;
52      num_active_writers: StatusN -> Int;
53      went_wrong: StatusN # Nat -> Bool;
54
55  % definitions for StatusB update functions
56  var
57      b: ArrayB;
58      w: Int;
59      s, v: Bool;
60      id: Nat;
61
62  eqn
63      default_statusB = statusB(all_true, 0, false);
64      start_reader(statusB(b, w, s), id) = statusB(set(b, id, w > 0), w, s);
65      w > 0 -> start_writer(statusB(b, w, s), id, v) =
66          statusB(all_true, w + 1, false);
67      w <= 0 -> start_writer(statusB(b, w, s), id, v) =
68          statusB(set(all_true, id, false), w + 1, v);
69      end_reader(statusB(b, w, s), id) = statusB(set(b, id, true), w, s);
70      end_writer(statusB(b, w, s), id) = statusB(set(b, id, true), w - 1, false);
71      stored_value(statusB(b, w, s)) = s;
72      num_active_writers(statusB(b, w, s)) = w;
73      went_wrong(statusB(b, w, s), id) = get(b, id);
74
75  % definitions for StatusN update functions
76  var
77      b: ArrayB;
78      w: Int;
79      s, v, id: Nat;
80
81  eqn
82      default_statusN = statusN(all_true, 0, 0);
83      start_reader(statusN(b, w, s), id) = statusN(set(b, id, w > 0), w, s);
84      w > 0 -> start_writer(statusN(b, w, s), id, v) =
85          statusN(all_true, w + 1, 0);
86      w <= 0 -> start_writer(statusN(b, w, s), id, v) =
87          statusN(set(all_true, id, false), w + 1, v);
88      end_reader(statusN(b, w, s), id) = statusN(set(b, id, true), w, s);
89      end_writer(statusN(b, w, s), id) = statusN(set(b, id, true), w - 1, 0);
90      stored_value(statusN(b, w, s)) = s;
91      num_active_writers(statusN(b, w, s)) = w;
92      went_wrong(statusN(b, w, s), id) = get(b, id);
93
94  % general functions.
95  % here, since the turn stores thread id's, we can use valid_id instead of in_domain
96  map
97      valid_id: Nat -> Bool;
98      other: Nat -> Nat;
99
100 var
101     n: Nat;
102
103 eqn
104     valid_id(n) = n < 2;
105     other(0) = 1;
106     other(1) = 0;
107
108 act
109     % indicator actions
110     wish, enter, leave: Nat;
111
112     % actions representing the register interactions
113     get_flag_start_r, get_flag_start_t, get_flag_start: Nat # Nat;
114     get_flag_end_r, get_flag_end_t, get_flag_end: Nat # Nat # Bool;
115     set_flag_start_r, set_flag_start_t, set_flag_start: Nat # Nat # Bool;
116     set_flag_end_r, set_flag_end_t, set_flag_end: Nat # Nat;
117     get_turn_start_r, get_turn_start_t, get_turn_start: Nat;
118     get_turn_end_r, get_turn_end_t, get_turn_end: Nat # Nat;
119     set_turn_start_r, set_turn_start_t, set_turn_start: Nat # Nat;
120     set_turn_end_r, set_turn_end_t, set_turn_end: Nat;
121
122 proc
123     % Flag and Turn registers
124     Flag(id: Nat, b: Bool, stat: StatusB) =
125         sum tid: Nat. valid_id(tid) -> (
126         sum b': Bool. set_flag_start_r(tid, id, b'). Flag(stat=start_writer(stat, tid, b'))
127         +
128         get_flag_start_r(tid, id). Flag(stat=start_reader(stat, tid))
```

```
129          +
130          (went_wrong(stat, tid) -> (
131              sum b': Bool. (
132              set_flag_end_r(tid, id). Flag(b=b', stat=end_writer(stat, tid))
133              +
134              get_flag_end_r(tid, id, b'). Flag(stat=end_reader(stat, tid))))
135          <> (
136              set_flag_end_r(tid, id). Flag(b=stored_value(stat), stat=end_writer(stat, tid))
137              +
138              get_flag_end_r(tid, id, b). Flag(stat=end_reader(stat, tid))))
139      );
140
141      Turn(n: Nat, stat: StatusN) =
142          sum tid: Nat. valid_id(tid) -> (
143          sum n': Nat. valid_id(n') -> set_turn_start_r(tid, n').
144          Turn(stat=start_writer(stat, tid, n'))
145          +
146          get_turn_start_r(tid). Turn(stat=start_reader(stat, tid))
147          +
148          (went_wrong(stat, tid) -> (
149              sum n': Nat. valid_id(n') -> (
150              set_turn_end_r(tid). Turn(n=n', stat=end_writer(stat, tid))
151              +
152              get_turn_end_r(tid, n'). Turn(stat=end_reader(stat, tid))))
153          <> (
154              set_turn_end_r(tid). Turn(n=stored_value(stat), stat=end_writer(stat, tid))
155              +
156              get_turn_end_r(tid, n). Turn(stat=end_reader(stat, tid))))
157      );
158
159      % the processes representing the algorithms
160      Thread(id: Nat) =
161          set_flag_start_t(id, id, true). set_flag_end_t(id, id)|wish(id).
162          set_turn_start_t(id, id). set_turn_end_t(id).
163          BusyWait(id);
164
165      BusyWait(id: Nat) =
166          get_flag_start_t(id, other(id)).
167          sum flag_other: Bool. get_flag_end_t(id, other(id), flag_other).
168          get_turn_start_t(id).
169          sum turn: Nat. valid_id(turn) ->
170          get_turn_end_t(id, turn).
171          (!flag_other || turn != id) -> Critical(id) <> BusyWait(id);
172
173      Critical(id: Nat) =
174          enter(id). leave(id).
175          set_flag_start_t(id, id, false). set_flag_end_t(id, id).
176          Thread(id);
177
178  % initialization
179  init
180      allow({
181          get_flag_start, get_flag_end, set_flag_start, set_flag_end,
182          get_turn_start, get_turn_end, set_turn_start, set_turn_end,
183          set_flag_end|wish, enter, leave},
184      comm({
185          get_flag_start_r|get_flag_start_t -> get_flag_start,
186          get_flag_end_r|get_flag_end_t -> get_flag_end,
187          set_flag_start_r|set_flag_start_t -> set_flag_start,
188          set_flag_end_r|set_flag_end_t -> set_flag_end,
189          get_turn_start_r|get_turn_start_t -> get_turn_start,
190          get_turn_end_r|get_turn_end_t -> get_turn_end,
191          set_turn_start_r|set_turn_start_t -> set_turn_start,
192          set_turn_end_r|set_turn_end_t -> set_turn_end},
193      Thread(0) || Thread(1) ||
194      Flag(0,false, default_statusB) || Flag(1,false, default_statusB) ||
195      Turn(0, default_statusN)
196      ));
```

C Model of Aravind's for 2 Threads with Safe Registers

```
 9  % Arrays over Boolean values
10  sort
11      ArrayB = Nat -> Bool;
12
13  map
14      all_true: ArrayB;
```

```
15      set: ArrayB # Nat # Bool -> ArrayB;
16      get: ArrayB # Nat -> Bool;
17
18  var
19      a: ArrayB;
20      n: Nat;
21      v: Bool;
22
23  eqn
24      all_true(n) = true;
25      set(a, n, v) = a[n -> v];
26      get(a, n) = a(n);
27
28  % We have two status objects, depending on the type of the stored value
29  sort
30      StatusB = struct statusB(ArrayB, Int, Bool);
31      StatusN = struct statusN(ArrayB, Int, Nat);
32
33  map
34      default_statusB: StatusB;
35      start_reader: StatusB # Nat -> StatusB;
36      start_writer: StatusB # Nat # Bool -> StatusB;
37      end_reader: StatusB # Nat -> StatusB;
38      end_writer: StatusB # Nat -> StatusB;
39      stored_value: StatusB -> Bool;
40      num_active_writers: StatusB -> Int;
41      went_wrong: StatusB # Nat -> Bool;
42
43      default_statusN: StatusN;
44      start_reader: StatusN # Nat -> StatusN;
45      start_writer: StatusN # Nat # Nat -> StatusN;
46      end_reader: StatusN # Nat -> StatusN;
47      end_writer: StatusN # Nat -> StatusN;
48      stored_value: StatusN-> Nat;
49      num_active_writers: StatusN -> Int;
50      went_wrong: StatusN # Nat -> Bool;
51
52  % definitions for StatusB functions
53  var
54      b: ArrayB;
55      w: Int;
56      s, v: Bool;
57      id: Nat;
58
59  eqn
60      default_statusB = statusB(all_true, 0, false);
61      start_reader(statusB(b, w, s), id) = statusB(set(b, id, w > 0), w, s);
62      w > 0 -> start_writer(statusB(b, w, s), id, v) =
63          statusB(all_true, w + 1, false);
64      w <= 0 -> start_writer(statusB(b, w, s), id, v) =
65          statusB(set(all_true, id, false), w + 1, v);
66      end_reader(statusB(b, w, s), id) = statusB(set(b, id, true), w, s);
67      end_writer(statusB(b, w, s), id) = statusB(set(b, id, true), w - 1, false);
68      stored_value(statusB(b, w, s)) = s;
69      num_active_writers(statusB(b, w, s)) = w;
70      went_wrong(statusB(b, w, s), id) = get(b, id);
71
72  % definitions for StatusN functions
73  var
74      b: ArrayB;
75      w: Int;
76      s, v, id: Nat;
77
78  eqn
79      default_statusN = statusN(all_true, 0, 0);
80      start_reader(statusN(b, w, s), id) = statusN(set(b, id, w > 0), w, s);
81      w > 0 -> start_writer(statusN(b, w, s), id, v) =
82          statusN(all_true, w + 1, 0);
83      w <= 0 -> start_writer(statusN(b, w, s), id, v) =
84          statusN(set(all_true, id, false), w + 1, v);
85      end_reader(statusN(b, w, s), id) = statusN(set(b, id, true), w, s);
86      end_writer(statusN(b, w, s), id) = statusN(set(b, id, true), w - 1, 0);
87      stored_value(statusN(b, w, s)) = s;
88      num_active_writers(statusN(b, w, s)) = w;
89      went_wrong(statusN(b, w, s), id) = get(b, id);
90
91  % constant value, the maximum value the data register can take
92  map
93      LAST: Nat;
94  eqn
```

```
95      LAST = 3; % 2*number of threads - 1
96
97  % general functions
98  map
99      valid_id: Nat -> Bool;
100     in_domain: Nat -> Bool;
101     other: Nat -> Nat;
102
103 var
104     n: Nat;
105
106 eqn
107     valid_id(n) = n < 2; % we have id's 0 and 1
108     in_domain(n) = n <= LAST;
109     other(0) = 1;
110     other(1) = 0;
111
112 act
113     % indicator actions:
114     wish, enter, leave : Nat;
115
116     % register interaction actions:
117     get_flag_start_r, get_flag_start_p, get_flag_start: Nat # Nat;
118     get_flag_end_r, get_flag_end_p, get_flag_end: Nat # Nat # Bool;
119     set_flag_start_r, set_flag_start_p, set_flag_start: Nat # Nat # Bool;
120     set_flag_end_r, set_flag_end_p, set_flag_end: Nat # Nat;
121
122         get_stage_start_r, get_stage_start_p, get_stage_start: Nat # Nat;
123         get_stage_end_r, get_stage_end_p, get_stage_end: Nat # Nat # Bool;
124         set_stage_start_r, set_stage_start_p, set_stage_start: Nat # Nat # Bool;
125         set_stage_end_r, set_stage_end_p, set_stage_end: Nat # Nat;
126
127         get_date_start_r, get_date_start_p, get_date_start: Nat # Nat;
128         get_date_end_r, get_date_end_p, get_date_end: Nat # Nat # Nat;
129         set_date_start_r, set_date_start_p, set_date_start: Nat # Nat # Nat;
130         set_date_end_r, set_date_end_p, set_date_end: Nat # Nat;
131
132 proc
133     % Flag, Stage and Date registers
134     Flag(id: Nat, b: Bool, stat: StatusB) =
135         sum pid: Nat. valid_id(pid) -> (
136         sum b': Bool. set_flag_start_r(pid, id, b'). Flag(stat=start_writer(stat, pid, b'))
137         +
138         get_flag_start_r(pid, id). Flag(stat=start_reader(stat, pid))
139         +
140         (went_wrong(stat, pid) -> (
141             sum b': Bool. (
142             set_flag_end_r(pid, id). Flag(b=b', stat=end_writer(stat, pid))
143             +
144             get_flag_end_r(pid, id, b'). Flag(stat=end_reader(stat, pid))))
145         <> (
146             set_flag_end_r(pid, id). Flag(b=stored_value(stat), stat=end_writer(stat, pid))
147             +
148             get_flag_end_r(pid, id, b). Flag(stat=end_reader(stat, pid))))
149     );
150
151     Stage(id: Nat, b: Bool, stat: StatusB) =
152         sum pid: Nat. valid_id(pid) -> (
153         sum b': Bool. set_stage_start_r(pid, id, b'). Stage(stat=start_writer(stat, pid, b'))
154         +
155         get_stage_start_r(pid, id). Stage(stat=start_reader(stat, pid))
156         +
157         (went_wrong(stat, pid) -> (
158             sum b': Bool. (
159             set_stage_end_r(pid, id). Stage(b=b', stat=end_writer(stat, pid))
160             +
161             get_stage_end_r(pid, id, b'). Stage(stat=end_reader(stat, pid))))
162         <> (
163             set_stage_end_r(pid, id). Stage(b=stored_value(stat), stat=end_writer(stat, pid))
164             +
165             get_stage_end_r(pid, id, b). Stage(stat=end_reader(stat, pid))))
166     );
167
168     Date(id: Nat, d: Nat, stat: StatusN) =
169         sum pid: Nat. valid_id(pid) -> (
170         sum n: Nat. in_domain(n) -> (set_date_start_r(pid, id, n).
171         Date(stat=start_writer(stat, pid, n)))
172         +
173         get_date_start_r(pid, id). Date(stat=start_reader(stat, pid))
174         +
```

```
175          (went_wrong(stat, pid) -> (
176              sum n': Nat. in_domain(n') -> (
177              set_date_end_r(pid, id). Date(d=n', stat=end_writer(stat, pid))
178              +
179              get_date_end_r(pid, id, n'). Date(stat=end_reader(stat, pid))))
180          <> (
181              set_date_end_r(pid, id). Date(d=stored_value(stat), stat=end_writer(stat, pid))
182              +
183              get_date_end_r(pid, id, d). Date(stat=end_reader(stat, pid))))
184      );
185
186      % the processes representing the algorithm
187      P(id: Nat) =
188          set_flag_start_p(id, id, true). set_flag_end_p(id, id)|wish(id). R1(id);
189
190      R1(id: Nat) =
191          set_stage_start_p(id, id, false). set_stage_end_p(id, id). R2(id);
192
193      R2(id: Nat) =
194          get_flag_start_p(id, other(id)).
195          sum flag_other: Bool. get_flag_end_p(id, other(id), flag_other).
196          get_date_start_p(id, id). sum d: Nat. in_domain(d) -> get_date_end_p(id, id, d).
197          get_date_start_p(id, other(id)).
198          sum date_other: Nat. in_domain(date_other) ->
199          get_date_end_p(id, other(id), date_other).
200          ((flag_other == false || d < date_other) -> R3(id) <> R2(id));
201
202      R3(id: Nat) =
203          set_stage_start_p(id, id, true) . set_stage_end_p(id, id).
204          get_stage_start_p(id, other(id)).
205          sum stage_other: Bool. get_stage_end_p(id, other(id), stage_other).
206          (stage_other -> R1(id) <> C(id));
207
208      C(id: Nat) =
209          enter(id). leave(id).
210          get_date_start_p(id, id). sum d: Nat. in_domain(d) -> get_date_end_p(id, id, d).
211          get_date_start_p(id, other(id)).
212          sum date_other: Nat. in_domain(date_other) ->
213          get_date_end_p(id, other(id), date_other).
214          set_date_start_p(id, id, max(d, date_other) + 1). set_date_end_p(id, id).
215          get_date_start_p(id, id). sum d_res: Nat. in_domain(d_res) ->
216          get_date_end_p(id, id, d_res).
217          ((d_res >= LAST) -> (
218          set_date_start_p(id, id, id). set_date_end_p(id, id).
219          set_date_start_p(id, other(id), other(id)). set_date_end_p(id, other(id)).
220          CF(id)) <> CF(id));
221
222      CF(id: Nat) =
223          set_stage_start_p(id, id, false). set_stage_end_p(id, id).
224          set_flag_start_p(id, id, false). set_flag_end_p(id, id) . P(id);
225
226  % initialization,
227  % hide handles replacing many action labels with tau, for verification ease
228  % allow and comm together handle the communication
229  % then the threads and registers are initialized with their id's and initial values
230  init
231      hide({
232          get_stage_start, get_stage_end, set_stage_start, set_stage_end,
233          get_flag_start, get_flag_end, set_flag_start, set_flag_end,
234          get_date_start, get_date_end, set_date_start, set_date_end},
235      allow({
236          wish|set_flag_end, enter, leave,
237          get_stage_start, get_stage_end, set_stage_start, set_stage_end,
238          get_flag_start, get_flag_end, set_flag_start, set_flag_end,
239          get_date_start, get_date_end, set_date_start, set_date_end},
240      comm({
241          get_flag_start_r|get_flag_start_p -> get_flag_start,
242          get_flag_end_r|get_flag_end_p -> get_flag_end,
243          set_flag_start_r|set_flag_start_p -> set_flag_start,
244          set_flag_end_r|set_flag_end_p -> set_flag_end,
245          get_stage_start_r|get_stage_start_p -> get_stage_start,
246          get_stage_end_r|get_stage_end_p -> get_stage_end,
247          set_stage_start_r|set_stage_start_p -> set_stage_start,
248          set_stage_end_r|set_stage_end_p -> set_stage_end,
249          get_date_start_r|get_date_start_p -> get_date_start,
250          get_date_end_r|get_date_end_p -> get_date_end,
251          set_date_start_r|set_date_start_p -> set_date_start,
252          set_date_end_r|set_date_end_p -> set_date_end},
253      P(0) || P(1) ||
254      Flag(0, false, default_statusB) || Flag(1, false, default_statusB) ||
```

```
255       Stage(0, false, default_statusB) || Stage(1, false, default_statusB) ||
256       Date(0, 0, default_statusN) || Date(1, 1, default_statusN)
257       )));
```

D Model of Aravind's for 3 Threads with Safe Registers

Differences with the 2 thread version are highlighted in red.

```
 9   % Arrays over Boolean values
10   sort
11       ArrayB = Nat -> Bool;
12
13   map
14       default_arrayB: ArrayB;
15       all_true: ArrayB;
16       set: ArrayB # Nat # Bool -> ArrayB;
17       get: ArrayB # Nat -> Bool;
18
19   var
20       a: ArrayB;
21       n: Nat;
22       v: Bool;
23
24   eqn
25       default_arrayB(n) = false;
26       all_true(n) = true;
27       set(a, n, v) = a[n -> v];
28       get(a, n) = a(n);
29
30   % Two status objects, depending on the type of the stored value
31   sort
32       StatusB = struct statusB(ArrayB, Int, Bool);
33       StatusN = struct statusN(ArrayB, Int, Nat);
34
35   map
36       default_statusB: StatusB;
37       start_reader: StatusB # Nat -> StatusB;
38       start_writer: StatusB # Nat # Bool -> StatusB;
39       end_reader: StatusB # Nat -> StatusB;
40       end_writer: StatusB # Nat -> StatusB;
41       stored_value: StatusB -> Bool;
42       num_active_writers: StatusB -> Int;
43       went_wrong: StatusB # Nat -> Bool;
44
45       default_statusN: StatusN;
46       start_reader: StatusN # Nat -> StatusN;
47       start_writer: StatusN # Nat # Nat -> StatusN;
48       end_reader: StatusN # Nat -> StatusN;
49       end_writer: StatusN # Nat -> StatusN;
50       stored_value: StatusN-> Nat;
51       num_active_writers: StatusN -> Int;
52       went_wrong: StatusN # Nat -> Bool;
53
54   % Status B functions
55   var
56       b: ArrayB;
57       w: Int;
58       s, v: Bool;
59       id: Nat;
60
61   eqn
62       default_statusB = statusB(all_true, 0, false);
63       start_reader(statusB(b, w, s), id) = statusB(set(b, id, w > 0), w, s);
64       w > 0 -> start_writer(statusB(b, w, s), id, v) =
65           statusB(all_true, w + 1, false);
66       w <= 0 -> start_writer(statusB(b, w, s), id, v) =
67           statusB(set(all_true, id, false), w + 1, v);
68       end_reader(statusB(b, w, s), id) = statusB(set(b, id, true), w, s);
69       end_writer(statusB(b, w, s), id) = statusB(set(b, id, true), w - 1, false);
70       stored_value(statusB(b, w, s)) = s;
71       num_active_writers(statusB(b, w, s)) = w;
72       went_wrong(statusB(b, w, s), id) = get(b, id);
73
74   % Status N functions
75   var
76       b: ArrayB;
77       w: Int;
```

```
78      s, v, id: Nat;
79
80  eqn
81      default_statusN = statusN(all_true, 0, 0);
82      start_reader(statusN(b, w, s), id) = statusN(set(b, id, w > 0), w, s);
83      w > 0 -> start_writer(statusN(b, w, s), id, v) =
84          statusN(all_true, w + 1, 0);
85      w <= 0 -> start_writer(statusN(b, w, s), id, v) =
86          statusN(set(all_true, id, false), w + 1, v);
87      end_reader(statusN(b, w, s), id) = statusN(set(b, id, true), w, s);
88      end_writer(statusN(b, w, s), id) = statusN(set(b, id, true), w - 1, 0);
89      stored_value(statusN(b, w, s)) = s;
90      num_active_writers(statusN(b, w, s)) = w;
91      went_wrong(statusN(b, w, s), id) = get(b, id);
92
93  % constant value, the maximum value allowed by the register domain
94  map
95      LAST: Nat;
96
97  eqn
98      LAST = 5; % 2 * number of threads - 1
99
100 % general functions, need two "other" now
101 map
102     in_domain: Nat -> Bool;
103     valid_id: Nat -> Bool;
104     other_fst:  Nat -> Nat;
105     other_scd:  Nat -> Nat;
106
107 var
108     n: Nat;
109
110 eqn
111     in_domain(n) =  n <= LAST;
112     valid_id(n) = n < 3;
113     other_fst(0) = 1;
114     other_scd(0) = 2;
115     other_fst(1) = 0;
116     other_scd(1) = 2;
117     other_fst(2) = 0;
118     other_scd(2) = 1;
119
120 % max operation for 3 values
121 map
122     max_3:  Nat # Nat # Nat -> Nat;
123
124 var
125     a, b, c:  Nat;
126
127 eqn
128     max_3(a, b, c) = max(max(a,b),c);
129
130 act
131     % indicator actions:
132     wish, enter, leave : Nat;
133
134     % register interaction actions:
135     get_flag_start_r, get_flag_start_p, get_flag_start: Nat # Nat;
136     get_flag_end_r, get_flag_end_p, get_flag_end: Nat # Nat # Bool;
137     set_flag_start_r, set_flag_start_p, set_flag_start: Nat # Nat # Bool;
138     set_flag_end_r, set_flag_end_p, set_flag_end: Nat # Nat;
139
140     get_stage_start_r, get_stage_start_p, get_stage_start: Nat # Nat;
141     get_stage_end_r, get_stage_end_p, get_stage_end: Nat # Nat # Bool;
142     set_stage_start_r, set_stage_start_p, set_stage_start: Nat # Nat # Bool;
143     set_stage_end_r, set_stage_end_p, set_stage_end: Nat # Nat;
144
145     get_date_start_r, get_date_start_p, get_date_start: Nat # Nat;
146     get_date_end_r, get_date_end_p, get_date_end: Nat # Nat # Nat;
147     set_date_start_r, set_date_start_p, set_date_start: Nat # Nat # Nat;
148     set_date_end_r, set_date_end_p, set_date_end: Nat # Nat;
149
150 proc
151     % Flag, Stage and Date registers
152     Flag(id: Nat, b: Bool, stat: StatusB) =
153         sum pid: Nat. valid_id(pid) -> (
154         sum b': Bool. set_flag_start_r(pid, id, b'). Flag(stat=start_writer(stat, pid, b'))
155         +
156         get_flag_start_r(pid, id). Flag(stat=start_reader(stat, pid))
157         +
```

```
158             (went_wrong(stat, pid) -> (
159                 sum b': Bool. (
160                 set_flag_end_r(pid, id). Flag(b=b', stat=end_writer(stat, pid))
161                 +
162                 get_flag_end_r(pid, id, b'). Flag(stat=end_reader(stat, pid))))
163             <> (
164                 set_flag_end_r(pid, id). Flag(b=stored_value(stat), stat=end_writer(stat, pid))
165                 +
166                 get_flag_end_r(pid, id, b). Flag(stat=end_reader(stat, pid))))
167         );
168
169         Stage(id: Nat, b:Bool, stat:StatusB) =
170             sum pid: Nat. valid_id(pid) -> (
171             sum b': Bool. set_stage_start_r(pid, id, b'). Stage(stat=start_writer(stat, pid, b'))
172             +
173             get_stage_start_r(pid, id). Stage(stat=start_reader(stat, pid))
174             +
175             (went_wrong(stat, pid) -> (
176                 sum b': Bool. (
177                 set_stage_end_r(pid, id). Stage(b=b', stat=end_writer(stat, pid))
178                 +
179                 get_stage_end_r(pid, id, b'). Stage(stat=end_reader(stat, pid))))
180             <> (
181                 set_stage_end_r(pid, id). Stage(b=stored_value(stat), stat=end_writer(stat, pid))
182                 +
183                 get_stage_end_r(pid, id, b). Stage(stat=end_reader(stat, pid))))
184         );
185
186         Date(id: Nat, d: Nat, stat: StatusN) =
187             sum pid: Nat. valid_id(pid) -> (
188             sum n: Nat. in_domain(n) -> (set_date_start_r(pid, id, n).
189             Date(stat=start_writer(stat, pid, n)))
190             +
191             get_date_start_r(pid, id). Date(stat=start_reader(stat, pid))
192             +
193             (went_wrong(stat, pid) -> (
194                 sum n': Nat. in_domain(n') -> (
195                 set_date_end_r(pid, id). Date(d=n', stat=end_writer(stat, pid))
196                 +
197                 get_date_end_r(pid, id, n'). Date(stat=end_reader(stat, pid))))
198             <> (
199                 set_date_end_r(pid, id). Date(d=stored_value(stat), stat=end_writer(stat, pid))
200                 +
201                 get_date_end_r(pid, id, d). Date(stat=end_reader(stat, pid))))
202         );
203
204         % The processes representing the threads
205         P(id: Nat) =
206             set_flag_start_p(id, id, true) . wish(id)|set_flag_end_p(id, id) . R1(id);
207
208         R1(id: Nat) =
209             set_stage_start_p(id, id, false) . set_stage_end_p(id, id) . R2(id);
210
211         R2(id: Nat) =
212             get_flag_start_p(id, other_fst(id)).  sum flag_fst:  Bool.
213             get_flag_end_p(id, other_fst(id), flag_fst).
214             get_flag_start_p(id, other_scd(id)).  sum flag_scd:  Bool.
215             get_flag_end_p(id, other_scd(id), flag_scd).
216             get_date_start_p(id, id). sum d: Nat. in_domain(d) ->
217             get_date_end_p(id, id, d).
218             get_date_start_p(id, other_fst(id)).  sum date_fst:  Nat.  in_domain(date_fst) ->
219             get_date_end_p(id, other_fst(id), date_fst).
220             get_date_start_p(id, other_scd(id)).  sum date_scd:  Nat.  in_domain(date_scd) ->
221             get_date_end_p(id, other_scd(id), date_scd).
222             (((flag_fst == false || d < date_fst) && (flag_scd == false || d < date_scd)) ->
223             R3(id) <> R2(id));
224
225         R3(id: Nat) =
226             set_stage_start_p(id, id, true) . set_stage_end_p(id, id).
227             get_stage_start_p(id, other_fst(id)).  sum stage_fst:  Bool.
228             get_stage_end_p(id, other_fst(id), stage_fst).
229             get_stage_start_p(id, other_scd(id)).  sum stage_scd:  Bool.
230             get_stage_end_p(id, other_scd(id), stage_scd).
231             ((stage_fst || stage_scd) ->
232             R1(id) <> C(id));
233
234         C(id: Nat) =
235             enter(id). leave(id).
236             get_date_start_p(id, id). sum d: Nat. in_domain(d) ->
237             get_date_end_p(id, id, d).
```

```
238           get_date_start_p(id, other_fst(id)).  sum date_fst:  Nat.  in_domain(date_fst) ->
239           get_date_end_p(id, other_fst(id), date_fst).
240           get_date_start_p(id, other_scd(id)).  sum date_scd:  Nat.  in_domain(date_scd) ->
241           get_date_end_p(id, other_scd(id), date_scd).
242           set_date_start_p(id, id, max_3(d, date_fst, date_scd) + 1). set_date_end_p(id, id).
243           get_date_start_p(id, id). sum d_res: Nat. in_domain(d_res) ->
244           get_date_end_p(id, id, d_res).
245           ((d_res >= LAST) -> (
246           set_date_start_p(id, id, id). set_date_end_p(id, id).
247           set_date_start_p(id, other_fst(id), other_fst(id)).  set_date_end_p(id, other_fst(id)).
248           set_date_start_p(id, other_scd(id), other_scd(id)).  set_date_end_p(id, other_scd(id)).
249           CF(id)) <> CF(id));
250
251       CF(id: Nat) =
252           set_stage_start_p(id, id, false). set_stage_end_p(id, id).
253           set_flag_start_p(id, id, false). set_flag_end_p(id, id). P(id);
254
255   % Initialization
256   init
257       hide({
258           get_stage_start, get_stage_end, set_stage_start, set_stage_end,
259           get_flag_start, get_flag_end, set_flag_start, set_flag_end,
260           get_date_start, get_date_end, set_date_start, set_date_end},
261       allow({
262           wish|set_flag_end, enter, leave,
263           get_stage_start, get_stage_end, set_stage_start, set_stage_end,
264           get_flag_start, get_flag_end, set_flag_start, set_flag_end,
265           get_date_start, get_date_end, set_date_start, set_date_end},
266       comm({
267           get_flag_start_r|get_flag_start_p -> get_flag_start,
268           get_flag_end_r|get_flag_end_p -> get_flag_end,
269           set_flag_start_r|set_flag_start_p -> set_flag_start,
270           set_flag_end_r|set_flag_end_p -> set_flag_end,
271           get_stage_start_r|get_stage_start_p -> get_stage_start,
272           get_stage_end_r|get_stage_end_p -> get_stage_end,
273           set_stage_start_r|set_stage_start_p -> set_stage_start,
274           set_stage_end_r|set_stage_end_p -> set_stage_end,
275           get_date_start_r|get_date_start_p -> get_date_start,
276           get_date_end_r|get_date_end_p -> get_date_end,
277           set_date_start_r|set_date_start_p -> set_date_start,
278           set_date_end_r|set_date_end_p -> set_date_end},
279       P(0) || P(1) || P(2) ||
280       Flag(0, false, default_statusB) ||
281       Flag(1, false, default_statusB) ||
282       Flag(2, false, default_statusB) ||
283       Stage(0, false, default_statusB) ||
284       Stage(1, false, default_statusB) ||
285       Stage(2, false, default_statusB) ||
286       Date(0, 0, default_statusN) ||
287       Date(1, 1, default_statusN) ||
288       Date(2, 2, default_statusN)
289       )));
```

E Model of Aravind's, Modified for Verification

In Section 8, we at various points introduced new actions for the verification of properties. This is the same model as shown in Appendix C, but with these new actions added in. For clarity, the modifications have been colored red. Note that depending on which property is being verified, it will differ which actions are hidden and allowed. We will first show the main model, and then the different initializations used for the properties discussed.

E.1 Main Model

```
9   % Arrays over Boolean values
10  sort
11      ArrayB = Nat -> Bool;
12
13  map
14      default_arrayB: ArrayB;
15      all_true: ArrayB;
16      set: ArrayB # Nat # Bool -> ArrayB;
17      get: ArrayB # Nat -> Bool;
18
```

```
19  var
20      a: ArrayB;
21      n: Nat;
22      v: Bool;
23
24  eqn
25      default_arrayB(n) = false;
26      all_true(n) = true;
27      set(a, n, v) = a[n -> v];
28      get(a, n) = a(n);
29
30  % Two status obkects, depending on the type of the stored value
31  sort
32      StatusB = struct statusB(ArrayB, Int, Bool);
33      StatusN = struct statusN(ArrayB, Int, Nat);
34
35  map
36      default_statusB: StatusB;
37      start_reader: StatusB # Nat -> StatusB;
38      start_writer: StatusB # Nat # Bool -> StatusB;
39      end_reader: StatusB # Nat -> StatusB;
40      end_writer: StatusB # Nat -> StatusB;
41      stored_value: StatusB -> Bool;
42      num_active_writers: StatusB -> Int;
43      went_wrong: StatusB # Nat -> Bool;
44
45      default_statusN: StatusN;
46      start_reader: StatusN # Nat -> StatusN;
47      start_writer: StatusN # Nat # Nat -> StatusN;
48      end_reader: StatusN # Nat -> StatusN;
49      end_writer: StatusN # Nat -> StatusN;
50      stored_value: StatusN-> Nat;
51      num_active_writers: StatusN -> Int;
52      went_wrong: StatusN # Nat -> Bool;
53
54  % Status B functions
55  var
56      b: ArrayB;
57      w: Int;
58      s, v: Bool;
59      id: Nat;
60
61  eqn
62      default_statusB = statusB(all_true, 0, false);
63      start_reader(statusB(b, w, s), id) = statusB(set(b, id, w > 0), w, s);
64      w > 0 -> start_writer(statusB(b, w, s), id, v) =
65          statusB(all_true, w + 1, false);
66      w <= 0 -> start_writer(statusB(b, w, s), id, v) =
67          statusB(set(all_true, id, false), w + 1, v);
68      end_reader(statusB(b, w, s), id) = statusB(set(b, id, true), w, s);
69      end_writer(statusB(b, w, s), id) = statusB(set(b, id, true), w - 1, false);
70      stored_value(statusB(b, w, s)) = s;
71      num_active_writers(statusB(b, w, s)) = w;
72      went_wrong(statusB(b, w, s), id) = get(b, id);
73
74  % Status N functions
75  var
76      b: ArrayB;
77      w: Int;
78      s, v, id: Nat;
79
80  eqn
81      default_statusN = statusN(all_true, 0, 0);
82      start_reader(statusN(b, w, s), id) = statusN(set(b, id, w > 0), w, s);
83      w > 0 -> start_writer(statusN(b, w, s), id, v) =
84          statusN(all_true, w + 1, 0);
85      w <= 0 -> start_writer(statusN(b, w, s), id, v) =
86          statusN(set(all_true, id, false), w + 1, v);
87      end_reader(statusN(b, w, s), id) = statusN(set(b, id, true), w, s);
88      end_writer(statusN(b, w, s), id) = statusN(set(b, id, true), w - 1, 0);
89      stored_value(statusN(b, w, s)) = s;
90      num_active_writers(statusN(b, w, s)) = w;
91      went_wrong(statusN(b, w, s), id) = get(b, id);
92
93  % constant value, the maximum value allowed by the register domain
94  map
95      LAST: Nat;
96  eqn
97      LAST = 3; % 2 * the number of threads - 1
98
```

```
% general functions
map
    valid_id: Nat -> Bool;
    in_domain: Nat -> Bool;
    other: Nat -> Nat;

var
    n: Nat;

eqn
    valid_id(n) = n < 2;
    in_domain(n) = n <= LAST;
    other(0) = 1;
    other(1) = 0;

act
    % labels for extra verification
    Loop;
    Reset:  Nat;
    TrueFlag:  Nat # Bool;
    Entered, SeenEntering:  Nat;
    TrueLeastRecentAccess:  Nat;

    % indicator actions:
    wish, enter, leave: Nat;

    % register interaction actions:
    get_flag_start_r, get_flag_start_p, get_flag_start: Nat # Nat;
    get_flag_end_r, get_flag_end_p, get_flag_end: Nat # Nat # Bool;
    set_flag_start_r, set_flag_start_p, set_flag_start: Nat # Nat # Bool;
    set_flag_end_r, set_flag_end_p, set_flag_end: Nat # Nat;

    get_stage_start_r, get_stage_start_p, get_stage_start: Nat # Nat;
    get_stage_end_r, get_stage_end_p, get_stage_end: Nat # Nat # Bool;
    set_stage_start_r, set_stage_start_p, set_stage_start: Nat # Nat # Bool;
    set_stage_end_r, set_stage_end_p, set_stage_end: Nat # Nat;

    get_date_start_r, get_date_start_p, get_date_start: Nat # Nat;
    get_date_end_r, get_date_end_p, get_date_end: Nat # Nat # Nat;
    set_date_start_r, set_date_start_p, set_date_start: Nat # Nat # Nat;
    set_date_end_r, set_date_end_p, set_date_end: Nat # Nat;

proc
    % Flag, Stage and Date registers
    Flag(id: Nat, b: Bool, stat: StatusB) =
        TrueFlag(id, b).  Flag() +
        sum pid: Nat. valid_id(pid) -> (
        sum b': Bool. set_flag_start_r(pid, id, b'). Flag(stat=start_writer(stat, pid, b'))
        +
        get_flag_start_r(pid, id). Flag(stat=start_reader(stat, pid))
        +
        (went_wrong(stat, pid) -> (
            sum b': Bool. (
            set_flag_end_r(pid, id). Flag(b=b', stat=end_writer(stat, pid))
            +
            get_flag_end_r(pid, id, b'). Flag(stat=end_reader(stat, pid))))
        <> (
            set_flag_end_r(pid, id). Flag(b=stored_value(stat), stat=end_writer(stat, pid))
            +
            get_flag_end_r(pid, id, b). Flag(stat=end_reader(stat, pid))))
    );

    Stage(id: Nat, b:Bool, stat:StatusB) =
        sum pid: Nat. valid_id(pid) -> (
        sum b': Bool. set_stage_start_r(pid, id, b'). Stage(stat=start_writer(stat, pid, b'))
        +
        get_stage_start_r(pid, id). Stage(stat=start_reader(stat, pid))
        +
        (went_wrong(stat, pid) -> (
            sum b': Bool. (
            set_stage_end_r(pid, id). Stage(b=b', stat=end_writer(stat, pid))
            +
            get_stage_end_r(pid, id, b'). Stage(stat=end_reader(stat, pid))))
        <> (
            set_stage_end_r(pid, id). Stage(b=stored_value(stat), stat=end_writer(stat, pid))
            +
            get_stage_end_r(pid, id, b). Stage(stat=end_reader(stat, pid))))
    );

    Date(id: Nat, d: Nat, stat: StatusN) =
```

```
179           sum pid: Nat. valid_id(pid) -> (
180           sum n: Nat. in_domain(n) -> (set_date_start_r(pid, id, n).
181           Date(stat=start_writer(stat, pid, n)))
182           +
183           get_date_start_r(pid, id). Date(stat=start_reader(stat, pid))
184           +
185           (went_wrong(stat, pid) -> (
186               sum n': Nat. in_domain(n') -> (
187               set_date_end_r(pid, id). Date(d=n', stat=end_writer(stat, pid))
188               +
189               get_date_end_r(pid, id, n'). Date(stat=end_reader(stat, pid))))
190           <> (
191               set_date_end_r(pid, id). Date(d=stored_value(stat), stat=end_writer(stat, pid))
192               +
193               get_date_end_r(pid, id, d). Date(stat=end_reader(stat, pid))))
194       );
195
196       % only works for the two thread example
197       LeastRecentAccess(v:  Nat) =
198           Entered(1).  LeastRecentAccess(0) +
199           Entered(0).  LeastRecentAccess(1) +
200           TrueLeastRecentAccess(v).  LeastRecentAccess();
201
202           % The actual processes
203       P(id: Nat) =
204           set_flag_start_p(id, id, true). set_flag_end_p(id, id)|wish(id) . R1(id);
205
206       R1(id: Nat) =
207           set_stage_start_p(id, id, false)|Loop. set_stage_end_p(id, id)|Loop. R2(id);
208
209       R2(id: Nat) =
210           get_flag_start_p(id, other(id))|Loop. sum flag_other: Bool.
211           get_flag_end_p(id, other(id), flag_other)|Loop.
212           get_date_start_p(id, id)|Loop. sum d: Nat. in_domain(d) ->
213           get_date_end_p(id, id, d)|Loop.
214           get_date_start_p(id, other(id))|Loop. sum date_other: Nat. in_domain(date_other) ->
215           get_date_end_p(id, other(id), date_other)|Loop.
216           ((flag_other == false || d < date_other) -> R3(id) <> R2(id));
217
218       R3(id: Nat) =
219           set_stage_start_p(id, id, true)|Loop. set_stage_end_p(id, id)|Loop.
220           get_stage_start_p(id, other(id))|Loop. sum stage_other: Bool.
221           get_stage_end_p(id, other(id), stage_other)|Loop.
222           (stage_other -> R1(id) <> C(id));
223
224       C(id: Nat) =
225           enter(id). leave(id).
226           get_date_start_p(id, id). sum d: Nat. in_domain(d) -> get_date_end_p(id, id, d).
227           get_date_start_p(id, other(id)). sum date_other: Nat. in_domain(date_other) ->
228           get_date_end_p(id, other(id), date_other).
229           set_date_start_p(id, id, max(d, date_other) + 1). set_date_end_p(id, id).
230           get_date_start_p(id, id). sum d_res: Nat. in_domain(d_res) ->
231           get_date_end_p(id, id, d_res).
232           ((d_res >= LAST) -> (
233           set_date_start_p(id, id, id). set_date_end_p(id, id)|Reset(id).
234           set_date_start_p(id, other(id), other(id)). set_date_end_p(id, other(id))|Reset(other(id)).
235           CF(id)) <> CF(id));
236
237       CF(id: Nat) =
238           set_stage_start_p(id, id, false). set_stage_end_p(id, id).
239           set_flag_start_p(id, id, false). set_flag_end_p(id, id).
240           P(id);
```

E.2 Initialization Property 3

```
 9  init
10      hide({
11          get_stage_start, get_stage_end, set_stage_start, set_stage_end,
12          get_flag_start, get_flag_end, set_flag_start, set_flag_end,
13          get_date_start, get_date_end, set_date_start, set_date_end,
14          TrueFlag, TrueLeastRecentAccess, Entered, Reset},
15      allow({
16          wish|set_flag_end,
17          enter, leave,
18          get_stage_start, get_stage_end, set_stage_start, set_stage_end,
19          get_flag_start, get_flag_end, set_flag_start, set_flag_end,
20          get_date_start, get_date_end, set_date_start, set_date_end,
21          get_stage_start|Loop, get_stage_end|Loop, set_stage_start|Loop, set_stage_end|Loop,
```

```
22          get_flag_start|Loop, get_flag_end|Loop, set_flag_start|Loop, set_flag_end|Loop,
23          get_date_start|Loop, get_date_end|Loop, set_date_start|Loop, set_date_end|Loop,
24          set_date_end|Reset}},
25      comm({
26          get_flag_start_r|get_flag_start_p -> get_flag_start,
27          get_flag_end_r|get_flag_end_p -> get_flag_end,
28          set_flag_start_r|set_flag_start_p -> set_flag_start,
29          set_flag_end_r|set_flag_end_p -> set_flag_end,
30          get_stage_start_r|get_stage_start_p -> get_stage_start,
31          get_stage_end_r|get_stage_end_p -> get_stage_end,
32          set_stage_start_r|set_stage_start_p -> set_stage_start,
33          set_stage_end_r|set_stage_end_p -> set_stage_end,
34          get_date_start_r|get_date_start_p -> get_date_start,
35          get_date_end_r|get_date_end_p -> get_date_end,
36          set_date_start_r|set_date_start_p -> set_date_start,
37          set_date_end_r|set_date_end_p -> set_date_end},
38      P(0) || P(1) ||
39      Flag(0, false, default_statusB) || Flag(1, false, default_statusB) ||
40      Stage(0, false, default_statusB) || Stage(1, false, default_statusB) ||
41      Date(0, 0, default_statusN) || Date(1, 1, default_statusN)
42      )));
```

E.3 Initialization Property 5

```
9   init
10      hide({
11          get_stage_start, get_stage_end, set_stage_start, set_stage_end,
12          get_flag_start, get_flag_end, set_flag_start, set_flag_end,
13          get_date_start, get_date_end, set_date_start, set_date_end,
14          TrueFlag, TrueLeastRecentAccess, Entered, Loop},
15      allow({
16          wish|set_flag_end,
17          enter, leave,
18          get_stage_start, get_stage_end, set_stage_start, set_stage_end,
19          get_flag_start, get_flag_end, set_flag_start, set_flag_end,
20          get_date_start, get_date_end, set_date_start, set_date_end,
21          get_stage_start|Loop, get_stage_end|Loop, set_stage_start|Loop, set_stage_end|Loop,
22          get_flag_start|Loop, get_flag_end|Loop, set_flag_start|Loop, set_flag_end|Loop,
23          get_date_start|Loop, get_date_end|Loop, set_date_start|Loop, set_date_end|Loop,
24          set_date_end|Reset}},
25      comm({
26          get_flag_start_r|get_flag_start_p -> get_flag_start,
27          get_flag_end_r|get_flag_end_p -> get_flag_end,
28          set_flag_start_r|set_flag_start_p -> set_flag_start,
29          set_flag_end_r|set_flag_end_p -> set_flag_end,
30          get_stage_start_r|get_stage_start_p -> get_stage_start,
31          get_stage_end_r|get_stage_end_p -> get_stage_end,
32          set_stage_start_r|set_stage_start_p -> set_stage_start,
33          set_stage_end_r|set_stage_end_p -> set_stage_end,
34          get_date_start_r|get_date_start_p -> get_date_start,
35          get_date_end_r|get_date_end_p -> get_date_end,
36          set_date_start_r|set_date_start_p -> set_date_start,
37          set_date_end_r|set_date_end_p -> set_date_end},
38      P(0) || P(1) ||
39      Flag(0, false, default_statusB) || Flag(1, false, default_statusB) ||
40      Stage(0, false, default_statusB) || Stage(1, false, default_statusB) ||
41      Date(0, 0, default_statusN) || Date(1, 1, default_statusN)
42      )));
```

E.4 Initialization Property 6

```
9   init
10      hide({
11          get_stage_start, get_stage_end, set_stage_start, set_stage_end,
12          get_flag_start, get_flag_end, set_flag_start, set_flag_end,
13          get_date_start, get_date_end, set_date_start, set_date_end,
14          Loop},
15      allow({
16          wish|set_flag_end,
17          leave, % note that enter is not allowed
18          get_stage_start, get_stage_end, set_stage_start, set_stage_end,
19          get_flag_start, get_flag_end, set_flag_start, set_flag_end,
20          get_date_start, get_date_end, set_date_start, set_date_end,
21          get_stage_start|Loop, get_stage_end|Loop, set_stage_start|Loop, set_stage_end|Loop,
22          get_flag_start|Loop, get_flag_end|Loop, set_flag_start|Loop, set_flag_end|Loop,
23          get_date_start|Loop, get_date_end|Loop, set_date_start|Loop, set_date_end|Loop,
```

```
24          TrueFlag, TrueLeastRecentAccess, SeenEntering,
25          set_date_end|Reset},
26      comm({
27          enter|Entered -> SeenEntering,
28          get_flag_start_r|get_flag_start_p -> get_flag_start,
29          get_flag_end_r|get_flag_end_p -> get_flag_end,
30          set_flag_start_r|set_flag_start_p -> set_flag_start,
31          set_flag_end_r|set_flag_end_p -> set_flag_end,
32          get_stage_start_r|get_stage_start_p -> get_stage_start,
33          get_stage_end_r|get_stage_end_p -> get_stage_end,
34          set_stage_start_r|set_stage_start_p -> set_stage_start,
35          set_stage_end_r|set_stage_end_p -> set_stage_end,
36          get_date_start_r|get_date_start_p -> get_date_start,
37          get_date_end_r|get_date_end_p -> get_date_end,
38          set_date_start_r|set_date_start_p -> set_date_start,
39          set_date_end_r|set_date_end_p -> set_date_end},
40      P(0) || P(1) ||
41      LeastRecentAccess(2) || %initialized to an unused id
42      Flag(0, false, default_statusB) || Flag(1, false, default_statusB) ||
43      Stage(0, false, default_statusB) || Stage(1, false, default_statusB) ||
44      Date(0, 0, default_statusN) || Date(1, 1, default_statusN)
45      )));
```

F mCRL2 Counterexamples

Here, we show and explain the mCRL2-generated counterexamples for the properties that we verified in Section 8 that did not hold on the model of Aravind's BLRU algorithm.

F.1 Counterexample Property 2

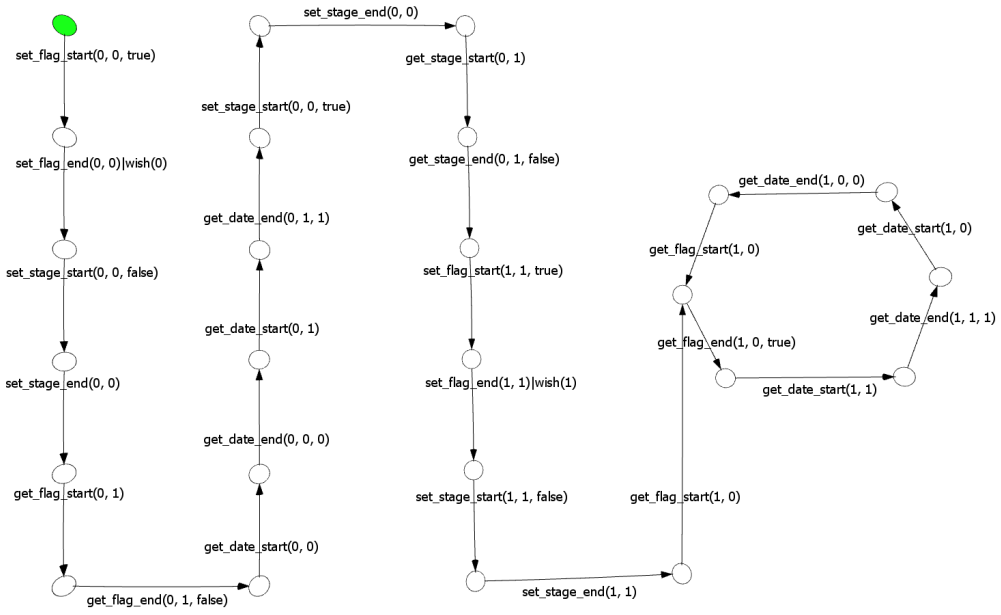

Figure 3: Counterexample for eventual access on Aravind's BLRU algorithm for 2 threads.

Figure 3 shows a counterexample for the eventual access property without fairness assumption. Here, we see that thread 0 starts competing. It gets as far as setting its own *stage* to *true* and seeing that the stage of thread 1 is still $false$, at this point thread 0 could execute the *enter* action. However, then thread 1 starts competing (at $set\_flag\_start(1, 1, true)$), it goes through the steps until it gets to the busy wait. There, since thread 0 still has its $flag$ set to *true* and the *date* of thread 1 is greater than that of thread 0, it cannot read the values that would allow it to pass the wait.

At this point, we would expect thread 0 to enter the critical section, and then execute its exit protocol, which would include making its own *date* greater than that of thread 1, which would free thread 1 from the busy wait. But if thread 0 never gets to take those actions, thread 1 can keep endlessly looping. This is what happens in this counterexample.

F.2 Counterexample Property 4 for 3 Threads

We found that a bound of 2 overtakes was not ensured when Aravind's algorithm is used with three threads. This was unsurprising, since the expected bound is $2n - 2$ where n is the number of threads. Still, the counterexample is included to show why the bound of 2 is not met. Figure 4 shows a counterexample generated by mCRL2. In this case, all actions except *wish*, *enter* and *leave* have been hidden and the example has been reduced modulo divergence-preserving branching bisimilarity.

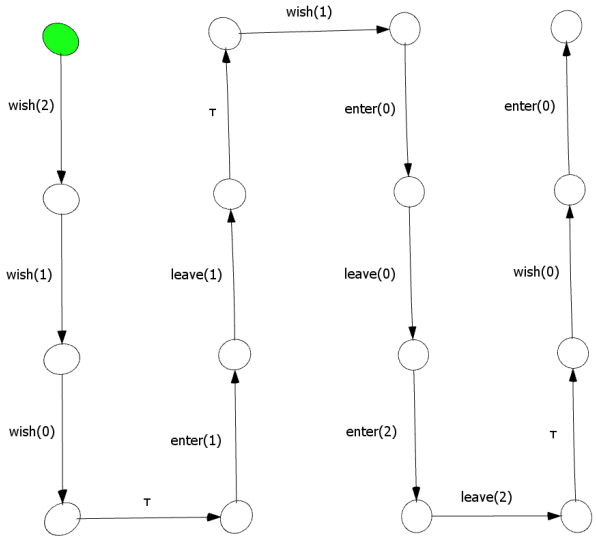

Figure 4: Counterexample for bounded overtaking with bound 2 on Aravind's BLRU algorithm for 3 threads.

We can still see what happens: all three threads raise their flag, and thread 1 manages to win the competition[3]. After thread 1 leaves the critical section and completes its exit protocol, the dates are as follows: $date[0] = 0, date[1] = 3, date[2] = 2$. Thread 1 wishes for access to the critical section again, the rest of the trace shows thread 1 being overtaken three times, violating the property.

First, thread 0 has the lowest *date* and enters the critical section, setting its own *date* to 4 in the exit protocol. Then thread 2 enters, since it now has the lowest *date*. In the exit protocol for thread 2, it sets its *date* to the maximum value, so a reset occurs. After the reset, $date[0] = 0, date[1] = 1, date[2] = 2$. When thread 0 then wishes for access to the critical section again, it can overtake thread 1 by having a lower *date* value. Thread 1 has been overtaken more than two times.

[3]Thread 1 gets access before thread 0 does even though thread 0 has a lower initial *date* value, we showed this is possible in the discussion of Property 6

F.3 Counterexample Property 6

Figure 5 shows a counterexample to Property 6. We are checking here that before a reset on the *date* values happens (and complicates LRU), the LRU property is satisfied. Here, we consider the LRU property satisfied if when both threads are competing, only the thread that had access to the critical section least recently can access it. This is a very strict definition of the LRU property.

The trace shown is rather long, so to summarize what happens: thread 0 goes through the whole cycle of doing the entry protocol, accessing the critical section, and then executing its exit protocol without thread 1 doing anything. The reason that this (minimal) counterexample includes this is because until one thread has had access to the critical section, we cannot say that either thread had access "least recently". It is only at the second $set\_flag\_start(0, 0, true)$ that the meat of the counterexample begins, this is near the top of the fourth vertical line. Once again, thread 0 goes through the motions of the entry protocol, and manages to get all the way to the end where it sets its own *stage* to *true* and sees that the stage of thread 1 is still $false$. At this point, thread 0 can take the *enter* action. But before it does, thread 1 raises its flag and starts competing. Then, thread 0 enters the critical section, even though thread 1 had access least recently and it currently competing.

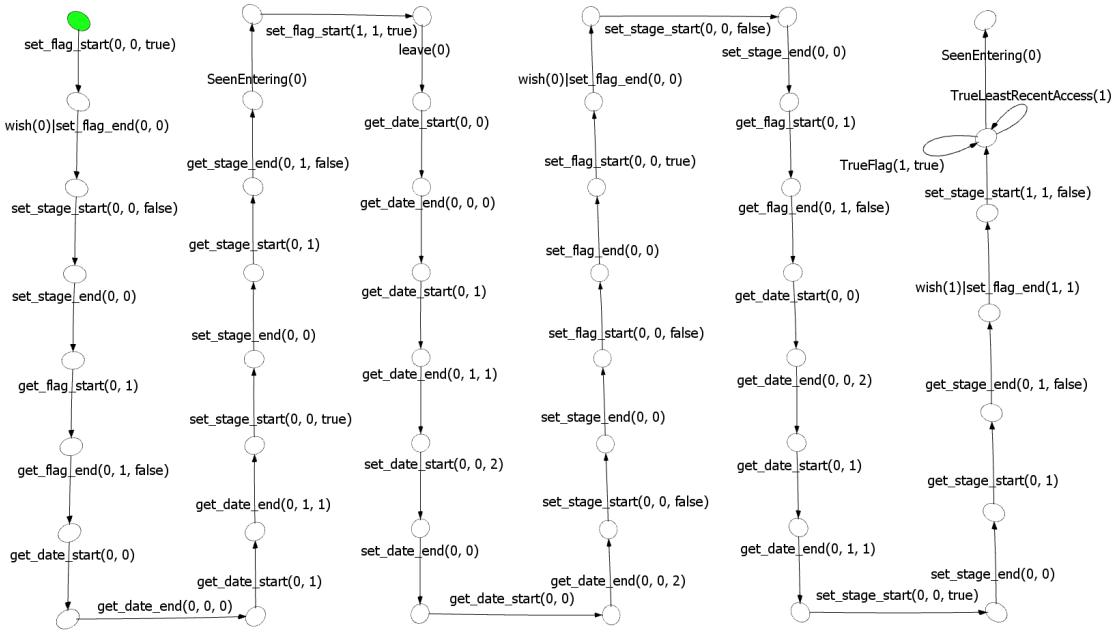

Figure 5: Counterexample for LRU on Aravind's BLRU algorithm for 2 threads, before a reset.