Tutorial: Designing Distributed Software in mCRL2

Preprint · April 2021

DOI: 10.48550/arXiv.2104.10542

2 authors, including:

Jan Friso Groote
Eindhoven University of Technology
283 PUBLICATIONS **6,534** CITATIONS

Tutorial: Designing Distributed Software in mCRL2

Jan Friso Groote[0000−0003−2196−6587] and Jeroen J.A. Keiren[0000−0002−5772−9527]

Department of Mathematics and Computer Science,
Eindhoven University of Technology, The Netherlands
{J.F.Groote,J.J.A.Keiren}@tue.nl

Abstract. Distributed software is very tricky to implement correctly as some errors only occur in peculiar situations. For such errors testing is not effective. Mathematically proving correctness is hard and time consuming, and therefore, it is rarely done. Fortunately, there is a technique in between, namely model checking, that, if applied with skill, is both efficient and able to find rare errors.

In this tutorial we show how to create behavioural models of parallel software, how to specify requirements using modal formulas, and how to verify these. For that we use the mCRL2 language and toolset (`www.mcrl2.org/`). We discuss the design of an evolution of well-known mutual exclusion protocols, and how model checking not only provides insight in their behaviour and correctness, but also guides their design.

Keywords: Model checking · Parallel software · Distributed software · mCRL2 toolset · Counterexamples

1 Introduction

Whoever designed parallel or distributed software and protocols must have found out how hard it is to get such software correct.[1] Distributed software defies testing, as some errors only occur very rarely, easily less than once in a million of runs. Yet, if such errors occur the software can go awry, with effects that range from confused internal administration, via crashing of the software, to loosing control over safety-critical hardware.

The theoretical solution is to prove the correctness, for instance using assertional methods that have been under development since the advent of the first electronic computers [1]. These days these methods are supported by proof checkers such as Coq [3] and Isabelle [22], or integrated automatic provers for algorithms such as Dafny [20]. These techniques are unprecedented in locating software faults and are unbeatable if it comes to delivering correct software.

[1] In this paper, for the sake of brevity, we generally refer to parallel or distributed software just using the term distributed software. The techniques discussed in this paper apply equally in both situations.

However, they have two important disadvantages. Proving the correctness of software can be very hard, as the proof may require tricky combinatorial arguments, and detailed bookkeeping. More importantly, it is very time consuming to provide a proof, even for a core algorithm, or a small distributed protocol. The net result of this is that proving correctness of actual software is hardly ever used in practical software development.

Fortunately, there is a method in between, namely model checking of models of the software. The idea is to use an abstract modelling language to model the essence of the distributed algorithm or protocol. Potential modelling languages with a powerful supporting model checking toolset are mCRL2 [14], LNT [10] and FDR3 [11] as they support behaviour with parallelism as well as all commonly used data types. Standard programming languages such as Java and C++ are less suitable for this purpose, as they are too versatile, and do not allow for concise mathematical formulation of protocols and algorithms. Domain Specific Languages to define automata based controllers such as ASD [19] and Dezyne [4] are suitable alternatives, with the advantage that they allow for code generation, but these languages generally provide limited verification possibilities.

Only formulating models of distributed algorithms already substantially improves the quality of a subsequent implementation. The reason is that models are more concise than implementations, and models tend to be studied and discussed more thoroughly than programs. Unfortunately, models still tend to contain errors and therefore, more needs to be done to increase the quality.

Improving the quality of software models further can be done by providing alternative independent views on the software and then comparing all views very precisely [6]. The probability to make the same mistake in all views is the product of the probabilities of making this mistake in each of the views. With a number of views the error probability drops dramatically, and error probabilities of 10^{-10} are attainable. In engineering such an approach is common where reliability is obtained due to redundancy. Even checking light-weight properties can already make a substantial difference [23].

We only take one alternative view, namely formulating compact properties on the model and proving them using model checking. Other views are making alternative models, independently making an implementation, specifying tests, and carrying out field tests. The more alternative views, the higher the quality of the result, provided they are very precisely compared to each other. Formulating correctness and proving this with a proof checker is also a valid alternative view.

We use the modal mu-calculus with data as it is unsurpassed in expressivity [14,5]. Fairness can be expressed using alternating fixed-points, and by using data complex behaviour of the model can be tracked and analysed with modal formulas. Alternative property languages, such as CTL/LTL can all be translated linearly to the modal mu-calculus with data [7].

In this tutorial we first describe mCRL2 and the modal mu-calculus very compactly. Subsequently, we focus on mutual exclusion protocols for shared memory and traverse through the development of such protocols, repeatedly identifying and repairing problems. We show how counterexamples that are very helpful in

identifying and understanding problems [26]. The modelling and analysis techniques described generalize to distributed algorithms in a straightforward way, using processes to model communication channels in stead of shared variables. We thus hope that this tutorial will help in understanding how to develop correct distributed algorithms and effectively obtain insight in their behaviour, which goes far beyond showing that they terminate with the right response.

2 mCRL2 primer

In this section we give a concise description of mCRL2 and the modal mu-calculus. More information is available in [14]. The language mCRL2 is based on process algebra [21,2]. The modal mu-calculus is based on Hennessy-Milner logic [18] and fixed point equations [5].

Process algebraic modelling centers around the notion of an action, typically denoted as `a`, `b`, `c`..., representing some atomic activity of a modelled entity, such as a program. Sending a message, writing a variable or printing some text are typical examples. If actions must happen at exactly the same time we denote them as multi-actions. By writing `a|b` it is indicated that actions `a` and `b` happen at the same instant in time. Actions and multi-actions have the same properties, and therefore we generally only speak about actions in the sequel.

Using the sequential composition operator (`.`) actions can be put in sequence and the choice operator (+) expresses that the behaviour of either the left or the right operand can be done. A typical example is `a.b+c.d` saying it is possible to do either an `a` followed by a `b` or a `c` followed by a `d`.

Processes are specified by recursive equations. The equation **`proc`** `P=a.P` indicates that the process `P` can infinitely often do an `a` action. Using **`init`** `P` it is expressed that process `P` is the behaviour defined by the specification.

Actions and processes can carry data, and all common data types are available. The process equation

`proc` `Adder(n:`*`Nat`*`)=`*`sum`* `m:`*`Nat`*`.add(m).Adder(n+m)`

is an example. The sum indicates the choice over all natural numbers. This process can perform one of the actions `add(m)` for every number `m` and continues with the behaviour `Adder(n+m)`. Behaviour can be executed conditionally on data using the if-then-else operator `b->p<>q` where `b` is a boolean expression and `p` and `q` are processes.

Processes are put in parallel using the parallel operator (`||`). Two parallel processes can communicate by synchronising their actions. This is denoted using the communication operator *`comm`*`({a_s|a_r->a},p||q)`, expressing that if action `a_s` and `a_r` can happen in `p` resp. `q`, these actions can happen together as `a`. We use the convention to write `_s` for send, and `_r` , after an action if they will be used for a communication. If actions `a_s` and `a_r` carry data they can only synchronise to `a` if the data in both actions are equal. Then `a` will have this data as parameter as well. To enforce that actions `a_s` and `a_r` must communicate, the allow operator is used. The process *`allow`*`({a},p)` expresses that only action `a` is allowed to happen in process `p` and all other actions are blocked.

The modal mu-calculus is an extension of propositional logic. Hence, we can use connectives such as `&&`, `||` and `!` representing *and*, *or* and *not*, respectively. Writing `<a>phi` expresses that an action `a` can be done after which `phi` holds, and `[a]phi` expresses that if an action `a` is done, then `phi` must hold afterwards. Instead of an action `a` we can use *`true`* to represent any action, and `!a` to represent any action but `a`. We can use a Kleene star to indicate arbitrary sequences of actions. So, `<!a*>phi` indicates that it is possible to do a sequence of actions in which `a` does not occur such that afterwards `phi` holds. The formula `[`*`true`*` *]phi` expresses that `phi` is valid after each sequence of actions. All actions can carry data, and quantification over data using *`exists`* and *`forall`* is possible.

Using the minimal fixed point operator *`mu`* `X.phi` and the maximal fixed point operator *`nu`* `X.phi` recurring properties can be specified. By *`nu`* `X.<a>X` we express that an infinite sequence of actions `a` must be possible. The formula *`mu`* `X.[!a]X&&<`*`true`*`>`*`true`* says that the action `a` must be done on every path within a finite number of actions. Using nested fixed points fairness properties can be expressed.

The fixed point variables can also use data. The following formula expresses that the total value offered to the adder will never exceed some maximum `M`:

```
nu X(n:Nat=0).forall m:Nat.[add(m)]X(n+m) &&
                [!exists m:Nat.add(m)]X(n) &&
                 val(n<M)
```

Here the variable `n`, initially equal to 0, sums up all values of `m` occuring in actions `add(m)`. The box modality with the exists expresses that whenever an action different from `add` is done, checking proceeds with an unaltered parameter `n`. Condition `n<M` guarantees that the sum `n` never exceeds `M`. Keyword **`val`** is needed to let the parser distinguish between modal formulas and data expressions.

3 Mutual exclusion

In this tutorial we study the mutual exclusion problem as we expect most of our readers to be familiar with it. This allows us to focus on how the mCRL2 toolset helps us to model and understand solutions for such a problem. The techniques we describe are equally applicable in other problem domains.

Dijkstra describes the mutual exclusion problem as follows [8]:

> "[...] consider N computers, each engaged in a process which, for our aims, can be regarded as cyclic. In each of the cycles a so-called 'critical section' occurs and the computers have to be programmed in such a way that at any moment only one of these N cyclic processes is in its critical section."

The first solution to the mutual exclusion problem has been known since 1959. It was first described by Dijkstra [9], who attributed it do Dekker. In this paper Dijkstra also show two simpler incorrect solutions. A first solution for N processes is due to Dijkstra [8] and only much later the well-known solution by Peterson appeared [25].

From Section 3.1 onward we model Dijkstra's algorithms in increasing complexity. Subsequently, we investigate Peterson's mutual exclusion algorithm.

Requirements. Before modelling solutions, we ask ourselves what the properties are that a mutual exclusion protocol should have. In order to understand the requirements it is necessary to understand that we model mutual exclusion using three phases. First, a *wish* is indicated to enter the critical section, second access is granted indicated by *enter*, after which the process indicates that is left the critical section using *leave*.

Mutual exclusion. At any moment only one of the processes is in its critical section.

Always eventually request. Every process can always eventually wish to enter its critical section.

Eventual access. Whenever a process indicates a wish to enter is critical section, it is guaranteed to eventually get access to its critical section. This property is also referred to as starvation freedom.

Bounded overtaking. There is an absolute bound B such that, whenever a process indicates it wants to enter its critical section, at most B processes can enter their critical section, before this process enters its critical section.

It is natural to formulate mutual exclusion as a property. But mutual exclusion is insufficient, as it can easily be guaranteed by never letting a process enter the critical section. For a properly functioning mutual exclusion protocol the second and third properties are equally important. The last one is interesting especially in systems where execution of programs does not need to be fair.

Memory model. We assume that the mutual exclusion protocols are implemented on a platform with shared memory where variables are written and read consecutively in some interleaved fashion by the parallel programs.

3.1 A naive algorithm for mutual exclusion

For two processes a naive solution of the mutual exclusion problem is Algorithm 1 suggested by Dijkstra [9]. It uses two global Boolean variables $flag[i]$ in which process i indicates that it is in its critical section. The algorithm, for process i now proceeds as follows. It first blocks until the flag of process $1-i$ becomes *false* using busy waiting. Once $flag[1-i]$ is *false*, the other process is not in its critical section. It then sets its own flag to *true* and enters its critical section. Once the work in the critical section is complete, it sets its flag to *false*.

Data: Global variables $flag[0], flag[1] : \mathbb{B}$
while $flag[1-i]$ **do** `/* Busy waiting */` **end**
$flag[i] := true$;
`/* Critical section */`
$flag[i] := false$;

Algorithm 1: A naive mutual exclusion algorithm for process i.

Below we go through a few steps to model this algorithm in mCRL2.

Shared variables. A shared variable can be modelled as process that carries the current value of the variable as a parameter. It can perform a read action, in which it sends the current value of its parameter. Also, it can perform a write action for each possible value that can be stored in the variable. The array *flag* is modelled by the following process.

```
proc Flag(i:Nat, b:Bool)=
       sum b':Bool. set_flag_r(i, b').Flag(i, b') +
       get_flag_s(i, b).Flag(i, b);
```

The name of the process is `Flag`. The parameter `i:Nat` describes the index in the array, and `b:Bool` gives the current value of the variable. Using `sum b':Bool.set_flag_r(i,b').Flag(i,b')` we model that the process can receive any new value `b'` from another process, and store it to parameter `b`. The action `get_flag_s(i, b).Flag(i, b)` allows to send the current value to any process that requests the value.

Modelling the busy waiting loop. The effect of the busy waiting loop is that the process can only continue when the guard becomes *false*, i.e., when $flag[1-i]$ has value *false*. We could model the busy waiting loop explicitly by a recursive process. However, in mCRL2, a read action blocks until the matching send action can also be performed. As the shared variable only sends its current value, we can model this loop by using `get_flag_r(1-i, false)`, that is, reading *false* from the flag of the other process. Since a subtraction results in an integer instead of a natural number, we need to add an explicit type conversion here, and write `get_flag_r(Int2Nat(1-i), false)`. As `i` is either 0 or 1, this is guaranteed to be natural number.

Model. We now combine this into an mCRL2 model. First, we define the behaviour of process i.

```
proc Mutex(i:Nat) =
       get_flag_r(Int2Nat(1-i), false).
       set_flag_s(i, true).
       enter(i).
       leave(i).
       set_flag_s(i, false).
       Mutex();
```

Note that the process is the sequential composition of the busy waiting loop, setting the flag of process i to *true* using `set_flag_s(i, true)`, entering the critical section using action `enter(i)`, leaving it using `leave(i)`, and setting the flag to *false* again. At the end of the algorithm we write `Mutex()` to model that the critical section can repeatedly be entered. Writing `Mutex()` without parameters is a shorthand that leaves the current value of the parameters unchanged. Here it is thus equivalent to writing `Mutex(i)`.

The system as a whole consists of two instances of `Mutex` and two shared variables, synchronising on `get_flag` and `set_flag`.

```
init allow({enter, leave, get_flag, set_flag},
       comm({get_flag_r | get_flag_s -> get_flag,
             set_flag_r | set_flag_s -> set_flag},
        Mutex(0) || Mutex(1) || Flag(0,false) || Flag(1,false)));
```

Here, the operator *comm* specifies that, `get_flag_r` and `get_flag_s` can synchronise. The result is is named `get_flag`. It does the same for `set_flag_r` and `set_flag_s`. Writing *allow*{`enter, leave, get_flag, set_flag`} specifies that we are only interested in the result of the communication, essentially enforcing synchronisation. We also allow the actions `enter` and `leave` that are local to the processes, and hence do not participate in any synchronisation.

Verification. Now that we have a model of this first mutual exclusion algorithm, we focus on its correctness. How can we formalize the mutual exclusion property using the mu-calculus? Observe that we explicitly modelled entering and leaving the critical section. Process i is therefore in its critical section if it performed an `enter(i)` action, but has not yet done the corresponding `leave(i)`. Mutual exclusion is then violated if we see two `enter` actions without an intermediate `leave`. This is captured in the following mu-calculus formula.

```
[true*][exists i1:Nat.enter(i1)]
       [!(exists i2:Nat.leave(i2))*][exists i3:Nat.enter(i3)]false
```

This formula expresses that invariantly (`[true*]`), after a process enters its critical section (`[exists i1:Nat.enter(i1)]`), as long as no leave action happened (`[!(exists i2:Nat.leave(i2))*]`), another process is not allowed to enter its critical section (`[exists i3:Nat.enter(i3)]false`).

We entered the model and the property in `mcrl2ide`, which is mCRL2's IDE that supports most basic uses of the mCRL2 toolset. A screenshot is shown in Figure 1. By clicking the 'Verify' button (green triangle) of the mutual exclusion property, the tools will verify whether the property holds. In this case, it finds that the property is violated, and the 'Verify' button changes into a red 'C'. By clicking the red 'C', the tool shows a counterexample. In this case, the counterexample is the one shown in Figure 2.

This counterexample is a trace where two processes execute an `enter` action without in intermediate `leave`. If we check the counterexample, it is immediately clear what is going on. Both processes check simultaneously that the other process is not in its critical section, concluding they can proceed to their critical section. Then, process 1 sets its flag and enters its critical section, immediately followed by process 0.

3.2 Fixing the naive algorithm

The problem with the naive algorithm is that each process first checks if the other process is in its critical section, and then sets its own flag. If this is done simultaneously, the processes do not observe that the other process is entering the critical section at the same time. We could potentially resolve this issue by first setting the flag, expressing the intent to enter the critical section, and then only proceed into the critical section if the flag of the other process is false. The improved algorithm is shown in Algorithm 2. It also stems from [9, Fig. 2].

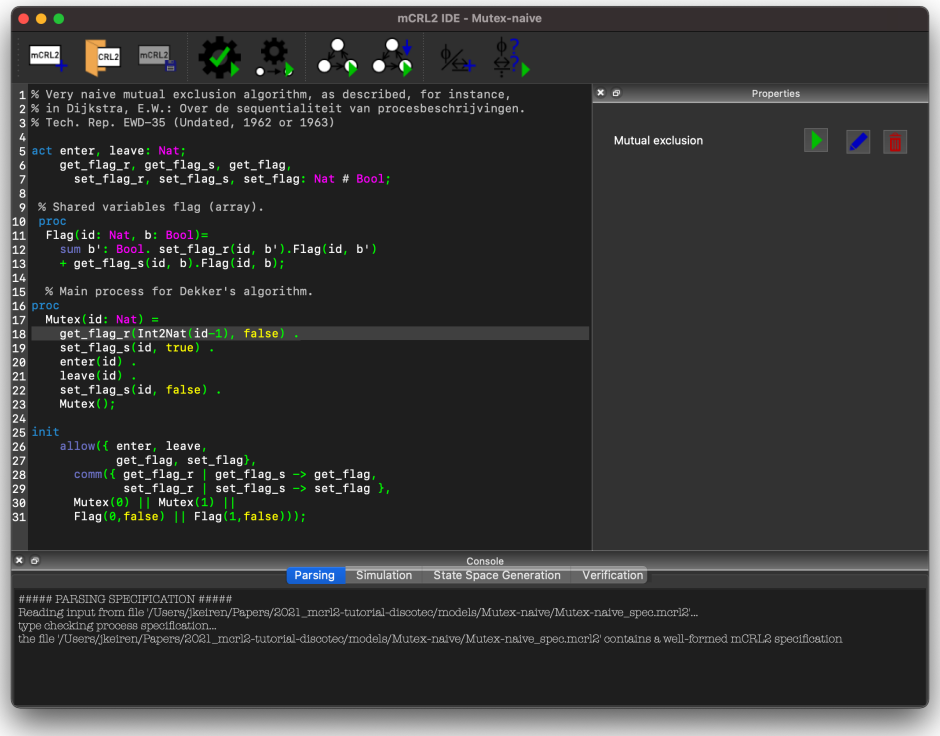

Fig. 1. Screenshot of mCRL2ide with naive mutual exclusion algorithm.

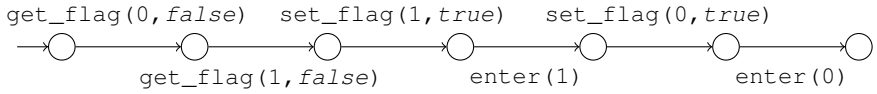

Fig. 2. Counterexample of the mutual exclusion property for the naive algorithm.

Data: Global variables $flag[0], flag[1] : \mathbb{B}$
$flag[i] := true$;
while $flag[1-i]$ **do** `/* Busy waiting */` **end**
`/* Critical section */`
$flag[i] := false$;

Algorithm 2: Improved naive mutual exclusion algorithm for process i.

Model. The change in the mCRL2 model is equally simple. We only exchange the first two lines of the `Mutex` process, which now becomes the following.

```
proc Mutex(i:Nat) =
     set_flag_s(i, true).
     get_flag_r(Int2Nat(1-i), false).
     enter(i).
     leave(i).
     set_flag_s(i, false).
     Mutex();
```

Verification. Changing the order of the program fixed the algorithm as it now satisfies the mutual exclusion property. So, we investigate the requirement that every process can always eventually wish to enter its critical section.

If process i sets its flag to *true* this means that it expresses the wish to enter its critical section. The property can then be expressed by saying that invariantly, for all processes i there is a path to a state in which process i can set its flag to *true*. This is expressed in the mu-calculus as follows.

```
[true*]forall i:Nat.val(i<=1) => <true*><set_flag(i, true)>true
```

Recall that `[true*]phi` is valid if `phi` holds in all reachable states. We express the property for all processes i using `forall i:Nat` with `val(i<=1)`. The remaining formula `<true*><set_flag(i, true)>true` expresses that there is a path to a state in which `set_flag(i, true)` can happen.

When we verify this property, it turns out that it does not hold. The counterexample is shown in Figure 3.

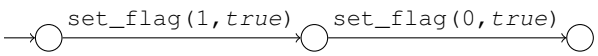

Fig. 3. Counterexample showing that a process cannot always eventually wish to enter.

This counterexample shows that if the processes simultaneously wish to enter the critical section, they block each other's possibility to proceed, disallowing both processes to set their own flag to *false*, leading to a deadlock. Due to the deadlock, they can never express their wish to reenter the critical section.

3.3 Dekker's algorithm

To resolve the deadlock in the previous mutual exclusion algorithm, Dekker's solution is to give priority to one of the two processes whenever both processes want to enter their critical section. We present it as Algorithm 3 [9].

Data: Global variables $flag[0], flag[1] : \mathbb{B}$ and $turn : \mathbb{N}$
$flag[i] := true$;
while $flag[1-i]$ **do**
  if $turn \neq i$ **then**
    $flag[i] := false$;
    while $flag[1-i]$ **do** `/* Busy waiting */` **end**
    $flag[i] := true$;
  end
end
$turn := 1-i$;
`/* Critical section */`
$flag[i] := false$;

Algorithm 3: Dekker's algorithm for process i.

The idea behind the algorithm is as follows. First, compared to the previous attempts, the meaning of global Boolean variables $flag[i]$ changes, and now indicates whether process i wishes to access its critical section. Second, a new shared variable *turn* indicates which process has priority when both processes want to enter their critical section.[2] The key idea now is that, while the other process $1-i$ wishes to enter its critical section, process i checks whether the other process has priority. If so, process i sets its flag to *false*, and then waits until process $1-i$ leaves its critical section and sets its flag to *false*. Then process i resets its flag to *true* and continues as before.

Model. To model this algorithm in mCRL2, we have to decide how to deal with the outer loop and the if-clause. We first discuss how to model the outer while-loop.

In more general terms, we want to model a program S_1; **while** b **do** S_2 **end**; S_3 in mCRL2. The most straightforward way to model this is to have two separate processes that are executed sequentially. The first process performs the behaviour of S_1 and hands execution over to the second process. The second process evaluates b. If b is *true* it executes the behaviour of S_2 and then executes itself, repeating the behaviour. Otherwise it executes the behaviour of S_3.

An if-then clause **if** b **then** S_1 **end**; S_2 can be modelled directly into the if-then-else construct `b->p<>q` of mCRL2. In this case `p` is the translation of $S_1; S_2$ and `q` is the translation of S_2. Note that both for the loop and the if-then clause, if the condition contains shared variables, their values must first be read.

The outer loop of the algorithm is modelled as follows.

```
proc Dekker_outer_loop(i:Nat) =
        sum flag_other:Bool.get_flag_r(other(i), flag_other).
        flag_other -> (sum turn: Nat.get_turn_r(turn).
                       (turn != i) -> (set_flag_s(i, false).
                                       get_turn_r(i).
                                       set_flag_s(i, true).
                                       Dekker_outer_loop(i)
                                      )
                                   <> Dekker_outer_loop(i)
                      )
                    <> (set_turn_s(other(i)).
                        enter(i).
                        leave(i).
                        set_flag_s(i, false).
                        Dekker(i);
                       );
```

Note that the shared variable `flag` of the other process is read, and its value is stored in `flag_other`. If the guard of the outer loop is true (`flag_other ->`), the loop is entered. In the body of the loop the turn variable is read, and it is decided whether the if-clause must be entered. In the body of the if the flag for this process is set to false, allowing the other process to enter its critical section. The process waits until the other process leaves its critical section. Note that here we use the construct we previously introduced for the busy waiting

[2] In [9], variables LA and LB are used as flags, and a Boolean variable AP is used in the place of *turn*.

loop. Subsequently, the flag of this process is set to true, and the while loop is repeated.

If the guard of the outer loop is false, we jump to the else part starting with the lower <> symbol, from where the rest of the process similar to our previous algorithms.

For the complete model, we also need a process modelling the global variable *turn*. This is done in a similar way as for the global array *flag*, where the variable is set and read using actions `set_turn` and `get_turn`, which are the results of synchronising `set_turn_r` and `set_turn_s`, and `get_turn_s` and `get_turn_r`. The parallel composition must be extended with the process modelling *turn*, as well as with an increased number of synchronising actions, and is given below. Some aspects of this process expression are explained in the next part on verification.

```
init allow({wish|set_flag, enter, leave,
            get_flag, set_flag, get_turn, set_turn},
       comm({get_flag_r | get_flag_s -> get_flag,
             set_flag_r | set_flag_s -> set_flag,
             get_turn_r | get_turn_s -> get_turn,
             set_turn_r | set_turn_s -> set_turn},
      Dekker(0) || Dekker(1) || Flag(0,false) || Flag(1,false) || Turn(0)));
```

Verification. As the algorithm keeps the same logic guarding the critical section as before, mutual exclusion is still satisfied. This is easily verified using modal formula given earlier.

However, to verify that we can always eventually request access to the critical section we need to be more careful. So far, we assumed that when a process sets its flag, this corresponds to expressing the wish to enter the critical section. However, as in Dekker's algorithm there are multiple places where the flag is set to true, we do not have this nice one-to-one correspondence. We therefore amend the model with an action `wish(i)` that makes the wish explicit the first time the process sets its flag. The main process therefore becomes the following.

```
proc Dekker(i:Nat) =
       wish(i)|set_flag_s(i, true).
       Dekker_outer_loop(i);
```

We here use a multi-action to model that `wish` and `set_flag` happen simultaneously. The set of allowed actions needs to be extended with `wish|set_flag`. We also need to modify the property to check for a such a multi-action instead of just the `set_flag`, hence the formula for always eventual request becomes the following.

```
[true*]forall i:Nat.val(i<=1) => <true*><wish(i)|set_flag(i, true)>true
```

This formula holds for Dekker's algorithm.

We now look at the property of eventual access. This says that, whenever a process wishes to enter its critical section, it inevitably ends up in the critical section. This can be formulated using the following mu-calculus formula.

```
[true*]forall i:Nat.val(i<=1)
      => [exists b:Bool.wish(i)|set_flag(i,b)]mu X.([!enter(i)]X && <true>true)
```

The formula says that invariantly, for every valid process `i`, when `i` wishes to enter its critical section (`[exists b:Bool.wish(i)|set_flag(i,b)]`), an `enter(i)` action inevitably happens within a finite number of steps (`mu X.([!enter(i)]X && <true>true)`). The conjunction `<true>true` ensures that that last part of the formula does not hold trivially in a deadlock state. Verifying the property yields false, and we get the counterexample shown in Figure 4.

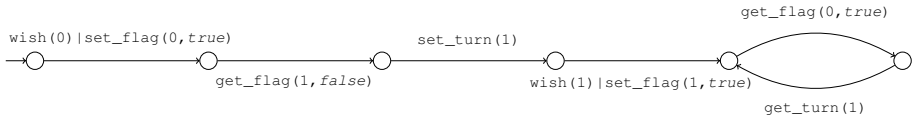

Fig. 4. Counterexample of the eventual access property for Dekker's algorithm.

The counterexample is interesting. It describes the scenario where process 0 requests access to its critical section, by setting the flag. It then checks the guard of the outer loop, which is false, and sets $turn := 1$ just before the critical section. Next, process 1 indicates it wants to access the critical section. Since $flag[0]$ is *true*, process 1 enters the outer loop, and since $turn = 1$, it will not enter into the if-statement, so it will keep cycling here until $flag[0]$ becomes *false*. What we see here is that, because process 1 is continuously cycling through the outer loop, process 0 never gets a chance to actually enter into its critical section. This is a typical fairness issue.

We could try to alter the formula in such a way that unfair paths such as in the counterexample satisfy the property, and are thus, essentially, ignored. In this case, we can do so by saying that each sequence not containing an `enter(i)` action ends in an infinite sequence of `get_flag` and `get_turn` actions. This results in the following formula.

```
[true*]forall i:Nat.val(i<=1) =>
   [exists b:Bool . wish(i)|set_flag(i,b)]
      nu X.mu Y.
         ([!enter(i) && !(exists i1:Nat.get_flag(i1, true) || get_turn(i1))]Y &&
          [exists i1:Nat.get_flag(i1, true) || get_turn(i1)]X)
```

Unfortunately, if we verify this property, we find it also does not hold. We get a different counterexample, which is shown in Figure 5.

What we see is that after process 1 wishes to enter its critical section, process 0 can come and enter the critical section infinitely many times, preventing process 1 from entering the critical section. A closer inspection reveals that this is because, to allow process 0 to enter, process 1 sets its flag to false, and then waits until process 0's flag becomes false. However, again, since we do not have any fairness guarantees, after setting its flag to false, process 0 can immediately request access to its critical section again, before process 1 observes that the flag became false.

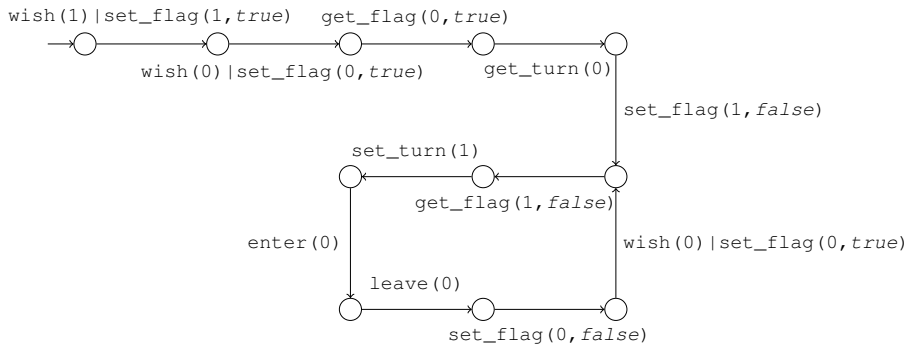

Fig. 5. Counterexample of the eventual access property under fairness for Dekker's algorithm.

We could, of course, try to change the property once more to exclude also this unfair execution. However, instead we we change our focus to Peterson's mutual exclusion protocol, as it is simpler, and therefore easier to analyse.

3.4 Peterson's mutual exclusion algorithm

Some of the issues in Dekker's algorithm, particularly regarding eventual access, are alleviated by Peterson's mutual exclusion protocol [25]. We previously presented a model of this algorithm in [12]. We describe this in Algorithm 3.4.

Data: Global variables $\mathit{flag}[0], \mathit{flag}[1] : \mathbb{B}$ and $\mathit{turn} : \mathbb{N}$
$\mathit{flag}[i] := \mathit{true}$;
$\mathit{turn} := 1-i$;
while $\mathit{flag}[1-i] \wedge \mathit{turn} = 1-i$ **do** `/* Busy waiting */` **end**
`/* Critical section */`
$\mathit{flag}[i] := \mathit{false}$;

Algorithm 4: Peterson's algorithm for process i.

In Algorithm 4, the *turn* variable is used differently from Dekker's algorithm. When a process requests access to the critical section by setting its flag, it will behave politely, and let the other process go first. It waits until either the other process does not ask for access to the critical section, i.e. $\mathit{flag}[1-i]$ is *false*, or the other process arrived later, in which case $\mathit{turn} = i$.

Model. Peterson's algorithm can be modelled in mCRL2 using the same principles we have used before. The structure of the initialization is completely analogous to that of the previous models. A single process executing Peterson's algorithm can be modelled as follows.

```
proc Process(i:Nat) =
        wish(i)|set_flag_s(i, true).
        set_turn_s(other(i)).
        (get_flag_r(other(i), false) + get_turn_r(i)).
        enter(i).
        leave(i).
        set_flag_s(i, false).
        Process(i);
```

Note that we use the fact that the negation of the guard of the loop is $\neg flag[1{-}i] \vee turn{=}i$, hence we can still use communicating actions to block until the guard becomes *false*.

Verification. This model satisfies all properties we investigated so far, including eventual access. This confirms the intuition we presented when introducing the algorithm. Let us now switch our attention to bounded overtaking, which we have not investigated yet.

Bounded overtaking says that if one process indicates its wish to enter, other processes can at most enter the critical section `B` times before this process is allowed to enter. It can be expressed as follows.

```
[true*] forall i:Nat.[exists b:Bool.wish(i)|set_flag(i,b)]
                     (nu Y(n:Nat = 0).val(n<=B) &&
                                      [!(exists i1:Nat.enter(i1))]Y(n) &&
                                      [enter(other(i))]Y(n+1) )
```

In this formula, for all processes `i`, whenever process `i` wishes to enter its critical section, we start to count the number of times the other process enters its critical section using the parameter `n`. All actions other than `enter` maintain the current value. Meanwhile, the property asserts that `n<=B`, i.e., the bound is satisfied.

Intuitively, we may expect that whenever a process wishes to enter its critical section, the other process may enter once first. However, if we check bounded overtaking with `B`=1, we get the counterexample shown in Figure 6.

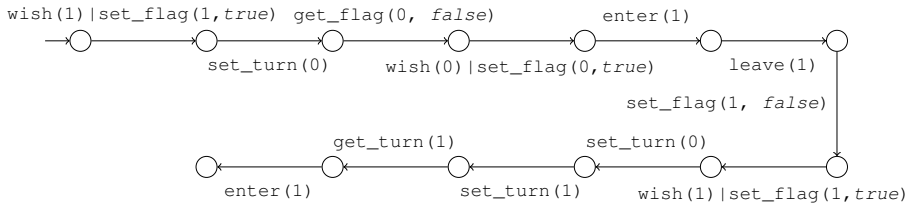

Fig. 6. Counterexample: Peterson does not satisfy bounded overtaking for $B{=}1$.

Let us take a close look at the counterexample. First, process 1 wishes to enter its critical section; it sets its flag, sets the turn to 0 and then checks the flag of process 0, which is currently *false*. At this point, process 1 is allowed to enter its critical section. However, before entering, process 0 also wishes to enter its critical section, and sets its flag. Subsequently, process 1 actually enters

the critical section, sets the turn to process 0, and only then process 0 sets the turn to 1, ultimately allowing process 1 to enter a second time. Hence, because process 0 is stalled after setting its flag, but before setting the turn to process 1, process 1 can overtake process 0 and enter a second time.

This leads to the question of whether overtaking for higher values of B is also possible. By reverifying the formula for B= 2, we find that the formula is valid. Bounded overtaking for Peterson's mutual exclusion protocol is limited to at most 1 times.

In [12] we investigated a version of Peterson's algorithm where, initially, one of the flags is set to *true* instead of *false*. This alternative initialisation was, at some point, described on Wikipedia [24]. It turns out that all four properties discussed above hold. However, if the flag of process 1 is initially *true*, process 0 will need the cooperation of process 1 to be allowed to enter for the first time, as it will set the turn to 1, and will block on the busy waiting loop.

This poses the question whether our properties are sufficient to cover the desired properties of mutual exclusion protocols. In particular one might want to verify the property that a process can always eventually request entry, without the other process having to perform any action. This is done by the following formula, which distinguishes Peterson's algorithm with and without correct initialisation.

```
[true*]forall i:Nat.val(i<=1) =>
          <!(wish(other(i))|set_flag(other(i), true)||
             set_turn(i)||
             get_flag(i, false)||
             get_turn(other(i))||
             enter(other(i))||
             leave(other(i))||
             set_flag(other(i),false))*><wish(i)|set_flag(i, true)>true
```

4 Epilogue

We went through several versions of mutual exclusion algorithms and showed that their correctness can be formulated and investigated using modal formulas. Although it requires skill and experience to write down process algebraic specifications, and in particular modal formulas with data, they provide a powerful pair of tools to investigate and design protocols and distributed algorithms. We used it to study and design many systems varying from games [16,15] to core protocols for embedded systems [17].

When the systems that are modelled become more complex, the state space grows, and verification of modal formulas becomes more time consuming, up to a point where the state space cannot be handled by contemporary tools. It turns out that the style of modelling has a substantial influence on how complex systems can become. In [13] 7 different specification guidelines are presented to keep the state space small.

References

1. Apt, K.R., Olderog, E.: Fifty years of hoare's logic. Formal Aspects Comput. **31**(6), 751–807 (2019). https://doi.org/10.1007/s00165-019-00501-3, https://doi.org/10.1007/s00165-019-00501-3
2. Bergstra, J.A., Klop, J.W.: The algebra of recursively defined processes and the algebra of regular processes. In: Paredaens, J. (ed.) Automata, Languages and Programming, 11th Colloquium, Antwerp, Belgium, July 16-20, 1984, Proceedings. Lecture Notes in Computer Science, vol. 172, pp. 82–94. Springer (1984). https://doi.org/10.1007/3-540-13345-3_7, https://doi.org/10.1007/3-540-13345-3_7
3. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions. Texts in Theoretical Computer Science. An EATCS Series, Springer (2004). https://doi.org/10.1007/978-3-662-07964-5, https://doi.org/10.1007/978-3-662-07964-5
4. van Beusekom, R., Groote, J.F., Hoogendijk, P.F., Howe, R., Wesselink, W., Wieringa, R., Willemse, T.A.C.: Formalising the dezyne modelling language in mcrl2. In: Petrucci, L., Seceleanu, C., Cavalcanti, A. (eds.) Critical Systems: Formal Methods and Automated Verification - Joint 22nd International Workshop on Formal Methods for Industrial Critical Systems - and - 17th International Workshop on Automated Verification of Critical Systems, FMICS-AVoCS 2017, Turin, Italy, September 18-20, 2017, Proceedings. Lecture Notes in Computer Science, vol. 10471, pp. 217–233. Springer (2017). https://doi.org/10.1007/978-3-319-67113-0_14, https://doi.org/10.1007/978-3-319-67113-0_14
5. Bradfield, J.C., Stirling, C.: Modal mu-calculi. In: Blackburn, P., van Benthem, J.F.A.K., Wolter, F. (eds.) Handbook of Modal Logic, Studies in logic and practical reasoning, vol. 3, pp. 721–756. North-Holland (2007). https://doi.org/10.1016/s1570-2464(07)80015-2, https://doi.org/10.1016/s1570-2464(07)80015-2
6. van den Brand, M., Groote, J.F.: Software engineering: Redundancy is key. Sci. Comput. Program. **97**, 75–81 (2015). https://doi.org/10.1016/j.scico.2013.11.020, https://doi.org/10.1016/j.scico.2013.11.020
7. Cranen, S., Groote, J.F., Reniers, M.A.: A linear translation from ctl* to the first-order modal μ -calculus. Theor. Comput. Sci. **412**(28), 3129–3139 (2011). https://doi.org/10.1016/j.tcs.2011.02.034, https://doi.org/10.1016/j.tcs.2011.02.034
8. Dijkstra, E.W.: Solution of a problem in concurrent programming control. Communications of the ACM **8**(9), 569 (Sep 1965). https://doi.org/10.1145/365559.365617
9. Dijkstra, E.W.: Over de sequentialiteit van procesbeschrijvingen (Undated, 1962 or 1963)
10. Garavel, H., Lang, F., Mateescu, R., Serwe, W.: CADP 2011: a toolbox for the construction and analysis of distributed processes. Int. J. Softw. Tools Technol. Transf. **15**(2), 89–107 (2013). https://doi.org/10.1007/s10009-012-0244-z, https://doi.org/10.1007/s10009-012-0244-z
11. Gibson-Robinson, T., Armstrong, P.J., Boulgakov, A., Roscoe, A.W.: FDR3: a parallel refinement checker for CSP. Int. J. Softw. Tools Technol. Transf. **18**(2), 149–167 (2016). https://doi.org/10.1007/s10009-015-0377-y, https://doi.org/10.1007/s10009-015-0377-y

12. Groote, J.F., Keiren, J.J.A., Luttik, B., de Vink, E.P., Willemse, T.A.C.: Modelling and Analysing Software in mCRL2. In: Arbab, F., Jongmans, S.S. (eds.) Formal Aspects of Component Software. pp. 25–48. Lecture Notes in Computer Science, Springer International Publishing, Cham (2020). https://doi.org/10.1007/978-3-030-40914-2_2
13. Groote, J.F., Kouters, T.W.D.M., Osaiweran, A.: Specification guidelines to avoid the state space explosion problem. Softw. Test. Verification Reliab. **25**(1), 4–33 (2015). https://doi.org/10.1002/stvr.1536, https://doi.org/10.1002/stvr.1536
14. Groote, J.F., Mousavi, M.R.: Modeling and Analysis of Communicating Systems. MIT Press (2014), https://mitpress.mit.edu/books/modeling-and-analysis-communicating-systems
15. Groote, J.F., de Vink, E.P.: Problem solving using process algebra considered insightful. In: Katoen, J., Langerak, R., Rensink, A. (eds.) ModelEd, TestEd, TrustEd - Essays Dedicated to Ed Brinksma on the Occasion of His 60th Birthday. Lecture Notes in Computer Science, vol. 10500, pp. 48–63. Springer (2017). https://doi.org/10.1007/978-3-319-68270-9_3, https://doi.org/10.1007/978-3-319-68270-9_3
16. Groote, J.F., Wiedijk, F., Zantema, H.: A probabilistic analysis of the game of the goose. SIAM Rev. **58**(1), 143–155 (2016). https://doi.org/10.1137/140983781, https://doi.org/10.1137/140983781
17. Groote, J.F., Willemse, T.A.C.: A symmetric protocol to establish service level agreements. Log. Methods Comput. Sci. **16**(3) (2020), https://lmcs.episciences.org/6812
18. Hennessy, M., Milner, R.: Algebraic laws for nondeterminism and concurrency. J. ACM **32**(1), 137–161 (1985). https://doi.org/10.1145/2455.2460, https://doi.org/10.1145/2455.2460
19. Hopcroft, P.J., Broadfoot, G.H.: Combining the box structure development method and CSP for software development. Electron. Notes Theor. Comput. Sci. **128**(6), 127–144 (2005). https://doi.org/10.1016/j.entcs.2005.04.008, https://doi.org/10.1016/j.entcs.2005.04.008
20. Leino, K.R.M., Wüstholz, V.: The dafny integrated development environment. In: Dubois, C., Giannakopoulou, D., Méry, D. (eds.) Proceedings 1st Workshop on Formal Integrated Development Environment, F-IDE 2014, Grenoble, France, April 6, 2014. EPTCS, vol. 149, pp. 3–15 (2014). https://doi.org/10.4204/EPTCS.149.2, https://doi.org/10.4204/EPTCS.149.2
21. Milner, R.: Communication and concurrency. PHI Series in computer science, Prentice Hall (1989)
22. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL - A Proof Assistant for Higher-Order Logic, Lecture Notes in Computer Science, vol. 2283. Springer (2002). https://doi.org/10.1007/3-540-45949-9, https://doi.org/10.1007/3-540-45949-9
23. Osaiweran, A., Schuts, M., Hooman, J.: Experiences with incorporating formal techniques into industrial practice. Empir. Softw. Eng. **19**(4), 1169–1194 (2014). https://doi.org/10.1007/s10664-013-9251-2, https://doi.org/10.1007/s10664-013-9251-2
24. Peterson's algorithm. https://en.wikipedia.org/wiki/Peterson%27s_algorithm (2015), accessed May 17
25. Peterson, G.L.: Myths about the mutual exclusion problem. Information Processing Letters **12**(3), 115–116 (Jun 1981). https://doi.org/10.1016/0020-0190(81)90106-X

26. Wesselink, W., Willemse, T.A.C.: Evidence extraction from parameterised boolean equation systems. In: Benzmüller, C., Otten, J. (eds.) Proceedings of the 3rd International Workshop on Automated Reasoning in Quantified Non-Classical Logics (ARQNL 2018) affiliated with the International Joint Conference on Automated Reasoning (IJCAR 2018), Oxford, UK, July 18, 2018. CEUR Workshop Proceedings, vol. 2095, pp. 86–100. CEUR-WS.org (2018), http://ceur-ws.org/Vol-2095/paper6.pdf