Modelling and analysing software in mCRL2

Technische Universiteit Eindhoven
Department of Mathematics and Computer Science

Modelling and Analysing Software in mCRL2

J.F. Groote, J.J.A. Keiren, B. Luttik, E.P. de Vink and T.A.C. Willemse

19/05

ISSN 0926-4515

editor: prof.dr.ir. J.J. van Wijk

Reports are available at:
https://research.tue.nl/en/publications/?search=Computer+science+reports+eindhoven&originalSearch=Computer+science+reports+eindhoven&pageSize=50&ordering=publicationYearThenTitle&descending=true&showAdvanced=false&allConcepts=true&inferConcepts=true&searchBy=RelatedConcepts

Computer Science Reports 19-05
Eindhoven, December 2019

Modelling and Analysing Software in mCRL2

J.F. Groote, J.J.A. Keiren, B. Luttik, E.P. de Vink, and T.A.C. Willemse

Faculty of Mathematics and Computer Science
Eindhoven University of Technology, The Netherlands
{J.F.Groote, J.J.A.Keiren, S.P.Luttik, E.P.d.Vink,
T.A.C.Willemse}@tue.nl

Abstract. Model checking is an effective way to design correct software. Making behavioural models of software, formulating correctness properties using modal formulas, and verifying these using finite state analysis techniques, is a very efficient way to obtain the required insight in the software. We illustrate this on four common but tricky examples.

1 Introduction

Software consists of algorithms that manipulate data structures, and protocols that describe how software components communicate. These algorithms and protocols are expected to work correctly under all the conditions they are designed for. However, it is not easy to foresee all possible situations the software will ever encounter. Corner cases and exceptional situations are typically difficult to identify upfront. Such situations are also hard to cover using testing.

A solution is to prove the correctness of the software. A wide range of methods, such as invariants, Hoare logics, separation logics, and process algebras, have been developed that allow this. Proof checkers like Coq [3] or Isabelle [22] allow to computer check such proofs, achieving an unparalleled level of trust in the quality of such proofs. Unfortunately constructing such proofs is still a manual, labour intensive activity.

Model checking and equivalence checking are two efficient verification techniques that strike a favourable balance between the ease of automation of testing and the level of trust established by correctness proofs. Both techniques rely on models made of the software under consideration, and they are particularly effective in exploring corner cases in software. In model checking a set of high-level requirements, phrased in terms of some temporal logic, are verified against the model. In equivalence checking, a state space is abstracted into a smaller one reflecting the essential properties of the original and which is sufficiently small to be inspected. In the related technique called refinement checking, a state space of the software is generated and compared to a required state space.

There are a number of toolsets that support one of these two approaches. For instance, the FDR [11] toolset specialises in refinement checking, and centres around the notion of failures-divergences refinement [25,21], which facilitates a step-wise refinement software development methodology. Toolsets such

as SPIN [15] and nuSMV [7] rely exclusively on model checking. CADP [10] and mCRL2 [6] are toolsets that offer both techniques.

In this paper we focus on the mCRL2 toolset and its three specification languages: the data language, which is based on the theory of abstract data types, the process specification language, which is an ACP-style process algebra, and the requirement language, which is a highly expressive extension of the modal μ-calculus. This toolset has been used successfully in a large number of case studies; see, *e.g.* [2,4,5,16,17,24], and it fulfils a role in education in various universities worldwide.

Our aim is to illustrate how to use mCRL2 to specify and analyse algorithms and data structures for which one should no longer wish to lull oneself into the belief that it does not warrant spending time on analysing their inner workings. The examples we consider, available via [1], are taken from the literature, and are presented such that they can immediately be replayed in the mCRL2 toolset.

We start by considering the well-known solution to the mutual exclusion problem, offered by Peterson's algorithm [23]. The interesting bit here is that—motivated by a non-standard exposition of the algorithm, present for some time on Wikipedia—we study the effects of different intialisations on the correctness of the algorithm. We furthermore study Knuth's dancing links [18], the concurrent data structure known as Treiber's stack [26] and Lamport's queue [20]. To facilitate a stand-alone exposition of the examples, we start with a brief primer to the languages used in mCRL2, where we assume that the reader is already familiar with similar-spirited languages.

With the current contribution, we hope to fill a void in the literature. As starting point to an uninitiated, motivated toolset user, (industrial) case studies can be discouraging due to their intrinsic complexity; on the other hand, the typical introductory problems often available in first-encounter tutorials[1] focus on language constructs and, for this reason, often lack in appeal. The aim of the current paper, therefore, is to bridge the gap between toy examples at one end, and complex case studies at the other end of the spectrum.

2 A short primer in mCRL2

In this section, we present a cursory overview of the most important language constructs in mCRL2. For a more elaborate introduction to the formalism, including a treatment of its real-time features, we refer to [12].

2.1 Data types

Most (distributed) algorithms and concurrent systems revolve around data in some form or another. The mCRL2 data language is based on higher-order abstract data types. This allows users to define their own data types, along with operations on these. For convenience, the mCRL2 data language also includes a large number of predefined standard data types and type constructors.

[1] See for instance the online mCRL2 introductory tutorial on `https://mcrl2.org/`

User defined data types. Abstract data types provide a straightforward mechanism for specifying complex data types. A user can declare new types, in this context called *sorts*, along with their (constructor) functions and their definitions, using a small number of primitives. Constructors are the atomic building blocks of a data type, allowing for an inductive definition of the type. A sort is declared using the **sort** keyword, whereas constructors are declared using the **cons** keyword; *e.g.*, for a given sort `A`, a declaration of an `A`-leaf tree could be:

```
sort Tree;
cons leaf: A -> Tree;
     node: Tree # Tree -> Tree;
```

Operators and functions that manipulate user-defined types can be declared using the **map** keyword. They are defined by a set of equations, introduced with the keyword **eqn**. These equations may refer to variables, which must be declared in a declaration block preceding the equations and announced by the keyword **var**. For most purposes, equations are interpreted as *rewrite rules*, allowing reasoning engines to manipulate and simplify (sub)expressions by matching the left-hand side of a rule and replacing these (sub)expressions by the right-hand side of a rule. For instance, assuming that `Max: A # A -> A` is a binary operation on sort `A`, a standard way of lifting that operator to a tree over `A`, given by function `Max_Leaf`, is as follows:

```
map Max_Leaf: T -> A;
var t1,t2: T; a: A;
eqn Max_Leaf(leaf(a)) = a;
    Max_Leaf(t1, t2) = Max(Max_Leaf(t1), Max_Leaf(t2));
```

Standard data types. For the convenience of the user, several standard data types and operations on these have been pre-defined.

One such standard data type, which, as we shall see later, is essential for the specification of conditional behaviour, is the sort *`Bool`*, representing the Booleans, with constructors *`false`* and *`true`*. Pre-defined operations on *`Bool`* include *negation* (denoted `!`), *conjunction* (denoted `&&`), *disjunction* (denoted `||`), and *implication* (denoted `=>`). A predicate on any datatype (user-defined or standard) can be defined as a mapping from that datatype to *`Bool`*. On all pre-defined datatypes binary equality and inequality predicates, respectively denoted == and `!=`, are defined with their standard interpretation. Furthermore, the language has generic constructions for universal quantification *`forall`* and existential quantification *`exists`*, which can be applied to predicates over any datatype. Most standard binary operators can be written *infix*.

In addition to the Booleans, the positive numbers *`Pos`*, natural numbers *`Nat`*, integers *`Int`* and reals *`Real`* are available, including many of the familiar operations on these. These numbers can be written in decimal notation; *e.g.*, the expression `10` represents the number ten. There is no pre-defined limitation on the size of the elements in these data types. Whether the tools can handle a specification involving numbers depends on the available computer memory, and so computations involving numbers should be done with care.

A further useful construct is that of an *enumerated type*, called *structured sort* in mCRL2. These structured sorts are a convenient way of defining sorts

with a finite set of elements, as they come with a built-in notion of equality. For instance, a data type `Colour` for the colours of a traffic light could be:

```
sort Colour = struct red | yellow | green;
```

Structured sorts are, however, more versatile than that. For instance, the `A`-leaf tree can alternatively be defined as follows:

```
sort Tree = struct leaf(A) | node(Tree, Tree);
```

The advantage of this definition over the one provided earlier, is that one does not need to bother defining equality as it is built-in for structured sorts.

Function types. The mCRL2 data language also has function types. An infinite list of natural numbers is a function from $\mathbb{N}$ to $\mathbb{N}$; in mCRL2 the data type representing this set of functions is the data sort `Nat -> Nat`. Functions can be defined using lambda abstraction, or by a pointwise specification of the result of applying the function to elements of its domain (function application works as expected). There is a concise mechanism for updating a function: assuming, for instance, that the function `id: Nat -> Nat` is the identity function, the function `id[3 -> 2]` represents the identity function in which the value `3` is mapped to `2`. Using lambda abstraction and function updates, operations such as removing the head of an existing list can, for instance, be modelled as follows.

```
map remove: (Nat -> Nat) -> (Nat -> Nat);
var l: Nat -> Nat;
eqn remove(l) = lambda n: Nat. l(n+1);
```

An alternative definition of the remove operation is as follows:

```
var l: Nat -> Nat; n: Nat;
eqn remove(l)(n) = l(n+1);
```

Type constructors. The mCRL2 data language has a number of useful type constructors. Lists, sets and bags can be defined in a generic way and come with pre-defined operations. For instance, the sort `List(A)` describes the data type of finite lists containing elements of type `A`, and comes with constructor `[]: List(A)` for the empty list, and (infix) `|>: A # List(A)-> List(A)` for prefixing a list. List *concatenation* is denoted by `++`, and further operators on lists include, *e.g.*, `head`, `rhead`, `tail`, `rtail`.

Sets can be described in a way that is close to standard notation. For instance, the sort `Set(Nat)` has all sets of all natural numbers as elements. The expression `{ n:Nat | n <= 10 }` describes the (finite) set of all numbers not exceeding ten, whereas `{ n:Nat | n > 10 }` describes its (infinite) complement. Set union, set difference, *etcetera*, are defined and work as expected. In a similar fashion one can define bags: `Bag(Nat)` describes the type of bags (multi-sets) over natural numbers.

Finally, sort aliasing can be used to give more meaningful names to data sorts. An abstraction on a set of identifiers is, *e.g.*, a natural number. However, one may prefer introducing a new data sort that is syntactically distinguishable from `Nat` to better reflect its role. This can easily be achieved as follows:

```
sort Id = Nat;
```

An aliased sort such as `Id` inherits all operations of the sort that it aliases.

2.2 Processes

Arguably, the most interesting aspect of a concurrent system is its behaviour. Behaviours can be represented by *Labelled Transition Systems* (LTSs); these are essentially directed, labelled graphs, where the vertices represent a system's state and the directed edges connecting two vertices are labelled with the event that causes the state change. A process algebra such as mCRL2 allows for specifying an LTS in a compositional fashion.

Sequential processes. A sequential process is a process that describes the possible behaviours of a system by way of actions (representing real-life or otherwise interesting events), combined sequentially, non-deterministically and using recursion. The process that cannot perform any activity (*i.e.*, is in a *deadlock*), is denoted `delta`. Somewhat different from other process algebras, the mCRL2 process algebra allows multiple actions to happen at the same time, resulting in a *multi-action*. Such a multi-action can be thought of as a multi-set of actions, all of which are assumed to happen simultaneously. The empty multi-action is denoted `tau`, whereas a multi-action of size one is often simply referred to as an action. An action may carry zero or more data arguments; this is useful for emiting relevant information of a process. Actions need to be declared explicitly using the **`act`** keyword:

```
act read, write: Nat;
```

Multi-actions can be constructed by listing the actions that are part of the multi-action; *e.g.*, `read(0)|read(0)|write(1)` denotes the multi-action consisting of two `read` actions, both with the same parameter, and one `write` action. Note that multi-action `a|b|tau|c` is equivalent to `a|b|c`.

Processes can be composed sequentially using a binary, associative sequential composition operator: process `p.q` denotes the process that first behaves as process `p` and, upon termination of `p`, continues to behave as process `q`. For instance, the process `write(0).read(0)` describes a process that first writes a value and subsequently reads a value.

Process `p + q` describes the process that chooses to behave as either process `p` or process `q`. The choice between the two processes is, in such a case, resolved by the first action that is executed: in case this action is due to process `p`, process `p` will dictate what behaviour is left, whereas when the first action is due to process `q`, process `q` will do so. In case one of the two processes cannot execute actions, the choice is automatically resolved in favour of the other process; *i.e.*, `p + delta` is behaviourally equivalent to `p`.

Note that in case the first action that is executed in `p + q` is offered both by `p` and `q`, the choice is resolved non-deterministically; that is, there is no guarantee which of the two processes will continue, and the choice cannot be

influenced. Such non-determinism is a powerful construct for modelling unreliabilities of a (sub)system and a useful mechanism for abstracting from decisions made internally by a (sub)system. For instance, in a process `write(0).read(0)+ write(0).read(-1)`, the execution of `write(0)` determines whether the same value is read as was written, or a different value; by only observing the write action, however, one cannot predict which of the two will happen.

Data can be used to influence the flow of control in a process by making process behaviour conditional: the ternary if-then-else construct `b -> p <> q` behaves as process `p`, provided that Boolean condition `b` holds true, and process `q` otherwise. For instance, the following process:

```
(reliable == true) -> write(0).read(0) <> write(0).read(-1)
```

behaves, perhaps, 'as expected' in case the Boolean `reliable` is true, and quirky otherwise. The binary if-then construct `b -> p` is short for `b -> p <> delta`.

Binary non-deterministic choice is generalised by non-deterministic choice quantification, which binds a 'local' variable. This is achieved by the construct `sum d:D. p`, in which variable `d` of data sort `D` is bound in the process expression `p`, and its value is chosen non-deterministically. Such a construct is useful to model, *e.g.*, a process that can write an arbitrary even value and then read it:

```
sum n: Nat. (n mod 2 == 0) -> write(n).read(n)
```

Note that, without further restrictions, this process yields an infinite state LTS; explicitly generating its transition system is therefore not feasible.

Infinite behaviours can be described using (parameterised) recursive equations, which associate behaviour to recursion variables (agents). The following process, for instance, describes the behaviour of a natural number buffer:

```
proc Buffer(b: Bool, n: Nat) =
  sum m: Nat. b -> write(m).Buffer(b = false, n = m) +
  !b -> read(n).Buffer(b = true);
```

In case parameter `b` is false, the value `n` that is currently stored in the buffer can be read through action `read(n)`. Otherwise, an arbitrary value can be written to the buffer. The role of parameter `b` in the above process is to indicate whether the buffer is empty or not. Note that, in recursive calls, only the parameters that change value need to be mentioned. Thus, in the first recursive call of `Buffer` the parameter `b` is set to `false` and `n` gets the value `m`, and in the second recursive call the parameter `b` is set to `true` and `n` remains unchanged.

Parallel processes. The parallel composition of processes `p` and `q` is denoted `p||q`. The action that a parallel composition `p||q` can execute can come either from process `p`, process `q`, or from processes `p` and `q` simultaneously. In the latter case, a multi-action consisting of an action from `p` and `q` is produced. Communication can be specified by a mapping that indicates which labels in a multi-action must synchronise; the trace of a successful communication is then a new action. Such a mapping is specified in a communication function through the keyword `comm`. This mapping renames multi-action labels to a new action label. A successful communication is subject to matching of the parameters of the individual actions in a multi-action. For instance, a process modelling a shared variable whose

value can be read through an action `value_s`, and another process continuously reading that value through an action `value_r`, may communicate to yield a new action with label `value`:

```
act value_s, value_r, value : Nat;
proc Variable(n: Nat) = value_s(n).Variable();
     Agent = sum m: Nat. value_r(m).Agent;
     Parallel = comm({ value_s|value_r -> value }, Variable(0) || Agent);
```

Process `Parallel` can execute an action `value_s(0)`, actions `value_r(0)`, `value_r(1)`, ..., and all multi-actions of the form `value_s(0)|value_r(v)`, where `v` is an arbitrary value not equal to `0`. In addition, the process can execute action `value(0)`. One is often only interested in the result of the communication and not in the individual actions that make up a communication. Actions can be 'filtered' using the *allow* operator:

```
proc Interact = allow({value}, Parallel);
```

The *allow* operator maps the (multi-)actions not explicitly listed to *delta*. There are several additional language constructs, such as *block* and *hide*; the former is, in a way, dual to the *allow* operator, whereas the latter maps a selected set of actions to the action *tau*, which is used for abstraction.

2.3 Modal formulas

The behaviour specified using the process and data languages can be analysed in order to determine whether it satisfies certain requirements. These requirements are specified in the first-order modal μ-calculus, a fixed point language based on a first-order extension of the modal logic called *Hennessy-Milner* logic (HML) [13]. The language offers the modal operators `<_>_` and `[_]_`, next to the familiar first-order logic constructs `||`, `&&`, *forall* and *exists*, and predicates **val**`(b)`, where `b` is an arbitrary Boolean expression from the data language. For a set of actions, represented by a formula `A`, the *may*-modality of the form `<A>f` holds true in a state whenever it allows for an action from the set `A` and leads to a state in which formula `f` holds true. The modal formula `[A]f` is the dual: a state satisfies this property when none of the actions from the set `A` lead to a state not satisfying `f`. For instance, process `Parallel` given above satisfies the properties `<value(0)>`*true* and `[value(1)]`*false*. The sets of actions used in modalities are also described using first-order logic, where, *e.g.*, `&&` denotes intersection, `||` denotes union, and *true* denotes the set of all actions. This way, one can write `[value(1)|| value(2)]`*false* to claim that neither value `1`, nor `2`, can be communicated. The property, `[`*exists* `n: Nat.` **val**`(n >= 1)&& value(n )]`*false*, or, equivalently, *forall* `n: Nat. [value(n)]`**val**`(n < 1)`, denotes that no value other than `0`, can be communicated.

Since HML is not capable of reasoning about the behaviours of unbounded depth, recursion is needed. This enters the language through a least fixed point operator *mu* `X. f` and a greatest fixed point operator *nu* `X. f`. Informally, a set of states satisfies *mu* `X. f` when each state satisfies some finite unfolding of `X`. For instance, the formula *mu* `X. (<a>X || <b>`*true*`)` indicates that there

should be some finite-depth formula <a>^{i}<b>*true* that is satisfied (although the depth i is potentially different for each state satisfying this formula). Dually, the formula *nu* X. ([a]X && [b]*false*) indicates that all possible unfoldings of the formula should hold. Essentially, this is the case when in no a-reachable state, a b-action is ever enabled.

Since fixed point formulas can be hard to understand, the mCRL2 requirement language offers regular expressions to reason about recursive processes. Using regular expressions, one can build a language from formulas describing sets of actions, sequential composition, choice and iteration. Such regular expressions can then be combined with the two modalities to yield expressions that permit to reason about processes of arbitrary or infinite depth. For example, the regular expression *true**.a represents the set of sequences consisting of zero or more arbitrary actions, followed by an action a. Consequently, formula [*true**.a]*false* asserts that none of these sequences are possible; *i.e.*, no a-action is ever possible. In a similar vein, formula <a*.b>*true* asserts that some sequence of a-actions leads to a state that can execute a b-action. Such regular formulas can be translated to standard fixed point formulas; *e.g.*, for a regular expression R and formula f, formula [R*]f is equivalent to *nu* X. ([R]X && f).

In mCRL2, fixed points can carry parameters, which can be useful to record information about the recursions that have been taken. For instance, formula *nu* X(n:*Nat* = 0). (**val**(n < Max)&& [a]X(n)&& [b]X(n+1)) only holds in case all a-b-runs of a system contain at most Max b-actions. This parameterisation is an incredibly powerful construct.

3 Using the mCRL2 toolset

The mCRL2 toolset has long been a collection of stand-alone tools, building on the philosophy that the user should be supported in using the transformations and solvers in as liberal a way as possible. While this philosophy has not been abandoned, the increasing popularity of the toolset has called for a more accessible way of using the toolset. For this reason, recent versions of the toolset come with a plain IDE, called mcrl2ide, which can be used to carry out basic analyses of mCRL2 specifications, without exposing the user to the overwhelming number of tools available in the toolset. The analyses for the examples we present in the next sections can be carried out from within this IDE. However, it should be noted that, for simplicity and accessibility, the IDE has opted to only implement a 'standard' workflow for all analyses, which may not always be optimal. In case non-standard, more advanced algorithms are needed for successfully carrying out an analysis, one needs to resort to the old way of working, using the file-driven environment mcrl2-gui or the command line.

4 Peterson's mutual exclusion algorithm

Peterson's mutual exclusion algorithm is a well-known protocol that coordinates different processes to obtain exclusive access to a shared resource by allowing at

most one process at a time to enter a critical section [23]. We focus on a setting with two processes, but the algorithm and our analysis generalise to any number of processes.

The algorithm uses three shared variables. For each of the two processes there is a shared Boolean variable *flag* which they set to *true* when they wish to enter the critical section. In addition, a variable *turn* is used to indicate which process is allowed to enter the critical section. Before entering, a process grants the other process access. If a process is granted access or the other process does not desire to enter the critical section, the critical section can be entered. In pseudo code the behaviour can be described as follows.

Global variables :	Behaviour of process i :
$flag[0]:\mathbb{B}$	$flag[i] := true$
$flag[1]:\mathbb{B}$	$turn := 1-i$
$turn:\mathbb{N}$	**while** $flag[1-i] = true \wedge turn = 1-i$ **do**
	busy wait
	critical section
	$flag[i] := false$

The initial value of the Boolean flags must be *false* for the algorithm to work correctly. However, different initialisations have appeared in online sources, cf. [27]; in such cases the behaviour is almost correct and the problem only surfaces by conducting a thorough analysis.

The mCRL2 model. To reason about the correctness of the algorithm in mCRL2, we introduce parameterised actions `wish`, `enter` and `leave` to model the interesting state changes of both processes. We remark that these actions are not needed for the correct functioning of the algorithm but they help in its analysis. Action `wish` signals a process' desire to enter the critical section. The action `enter` marks the moment a process enters the critical section and `leave` signals the process leaving the critical section. The assignments in the algorithm itself are modelled using actions `get_flag`, `set_flag`, `get_turn` and `set_turn` through which the shared variables can be read and set. The shared variables are, as remarked in Section 2, typically modelled as processes, and assignment to, and checks on these variables are modelled by the communications with these proceses. Peterson's algorithm, and our model by extension, uses only standard data structures. In our model, we identify each of the two processes by a number, and we use a custom mapping `other` to obtain the identity of the other process.

```
act wish, enter, leave: Nat;
    get_flag_r, get_flag_s, get_flag,
      set_flag_r, set_flag_s, set_flag: Nat # Bool;
    get_turn_r, get_turn_s, get_turn,
      set_turn_r,set_turn_s,set_turn: Nat;

map other: Nat -> Nat;
eqn other(0) = 1;
    other(1) = 0;
```

The processes `Flag(0,true)`, `Flag(1,true)` and `Turn(0)` depicted below model the three shared variables. The additional argument for these processes sets the relevant initial values.

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

  Turn(n:Nat)=
    sum n': Nat. set_turn_r(n').Turn(n') + get_turn_s(n).Turn(n);
```

A single thread of the algorithm is represented by `Process`. Modelling action `wish` and the `true`-assignment to the *flag* variable are specified to occur simultaneously, because the latter marks the wish of the process to enter the critical section. This highlights a typical use-case for multi-actions. Our model abstracts from the busy waiting loop in the algorithm by modelling the loop by a single communication. Using that actions synchronise on values, it is only necessary to check whether one of the shared variables attains a value that allows to enter the critical section.

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

The initialisation of the process is shown below. Note that we explicitly hide the communications with the shared variables; this allows for focussing on the interesting state changes of both processes. As a result, the only actions, apart from `tau`, that remain in the state space of the algorithm are the `enter`, `leave` and `wish` actions (the latter results from the underlying theory, stating that `wish(0)|tau` is the (multi-)action `wish(0)`).

```
init
  hide({ get_flag, set_flag, get_turn, set_turn },
    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 },
      Process(0) || Process(1) ||
      Flag(0,false) || Flag(1,false) || Turn(0))));
```

The analysis. There are three fundamental requirements a mutual exclusion algorithm must meet. The first one asserts mutual exclusion: at no time is it possible to be able to do two `enter` actions without a `leave` action in between.

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

The second property says that whenever a process wishes to enter, it is allowed access within a finite number of steps. A simple formulation is the following: whenever an action `wish(id)` happens, an action `enter(id)` is guaranteed to follow within a finite number of steps. The latter requires a least fixed point.

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

The third property is called bounded overtaking: whenever one process indicates the wish to enter the critical section, the other process can at most enter the critical section twice. This is formalised by asserting that after a `wish(id)`, at most two `enter(other(id))` actions can happen without an `enter(id)` somewhere in between. Counting of the number of occurrences of `enter(other(id))` actions is taken care of by the parameter `n` in the formula; each time such an action is encountered, `n` is increased, but in all cases, the formula asserts that it never invalidates condition **`val`**`(n<=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) )
```

The stronger property that if a process wishes to enter the critical section, the other process can enter the critical section only once is invalid.

The three properties above hold true for Peterson's algorithm. One may wonder whether an out-of-order execution, which is common in modern processors, affects the correctness of the algorithm. Out-of-order execution means that if a sequential process writes to unrelated memory addresses, writing can take place in any order. As the shared variables *flag* and *turn* are stored at different addresses, there is no guarantee that assignments are executed in the order as listed. It is easy to change the model and prove that it violates the mutual exclusion property, whereas the other two properties remain valid. This means that, to guarantee correctness on contemporary hardware, this algorithm requires special measures that prevent swapping certain instruction.

Remarkably, the validity of the three properties is independent of the initialisation of the algorithm. To gain some further understanding, we hide all actions, except `enter` and `leave` and apply weak trace minimisation. The corresponding labelled transition systems are depicted in Figure 1.

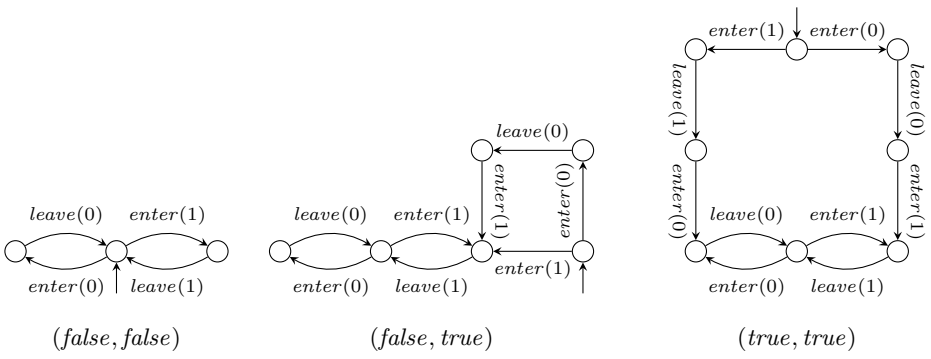

Fig. 1. The weak trace reduced behaviour of Peterson's algorithm with different initialisations.

At the left the graph with the correct initialisation $(false, false)$ of the flags is depicted. In the middle the initialisation $(false, true)$ and at the right $(true, true)$

can be found. The initial states have a small incoming arrow. When the initialisation is not correct, process 0, after it has entered the critical section once, is forced to wait until process 1 entered its critical section for the first time. For the transition system at the right, the reverse is also true.

The transition systems suggest that the second formula, expressing that a process that wishes to enter the critical section will be granted access after a finite number of steps, is insufficient, since it does not take into account that one process' capability of entering the critical section should not dependent on the other process' desire to do so. To take this into account, we phrase the following requirement: whenever a process expresses the wish to enter the critical section, it can enter the critical section on its own accord, unless the other process already expressed a wish to enter the critical section, enters the critical section or expresses the wish to enter the critical section. We again use a counter `n` to record the number of processes that have expressed the wish to enter the critical section and have not yet left the critical section. In case `n==0`, we use a least fixed point to check that for process `id` wishing to enter the critical section, along all paths not involving `wish` or `enter` actions of process `other(id)`, process `id` inevitably enters the critical section.

```
nu X(n: Nat = 0).
  ([exists id: Nat. wish(id)]X(n+1) && [exists id: Nat. leave(id)]X(max(0,n-1))) &&
  [!exists id: Nat. wish(id) || leave(id)]X(n) &&
  (val(n==0) => forall id:Nat. [wish(id)]
    ( mu Y. ([!enter(id) && !(wish(other(id)) || enter(other(id)))]Y &&
        <!(wish(other(id)) || enter(other(id)))> true)))
```

It turns out that this formula distinguishes between Peterson's mutual exclusion algorithm with and without proper initialisation.

5 Knuth's Dancing Links

Dancing links [14,18] is a technique to efficiently perform removal and insertion operations on a circular doubly linked list. It is intended to be used when elements temporarily need to be removed from the list in the course of a computation, and have to be re-inserted at the same position at a later stage. Knuth used the technique in his *Algorithm X*, which solves the exact cover problem. The correctness claim for the operations is that whenever a sequence of removals of elements $x_0, \ldots, x_n$ is applied, followed by a sequence of insertions of elements $x_n, \ldots, x_0$, then the result is again the original list. A verification of this property was a challenge of the VerifyThis [9] competition in 2015. We include a verification of the dancing links technique to illustrate how mCRL2 can be used to verify the correctness of operations on data structures.

Let x be an element of a circular doubly linked list, and suppose that $L(x)$ refers to its predecessor and $R(x)$ to its successor. The operations $\mathit{remove}(x)$ and $\mathit{insert}(x)$ are defined by

$$\begin{aligned} &\mathit{remove}(x) : L(R(x)) := L(x);\ R(L(x)) := R(x), \text{ and} \\ &\mathit{insert}(x) :\ \ L(R(x)) := x;\ R(L(x)) := x. \end{aligned}$$

The idea is that, after removal, the element x is not garbage-collected, waiting to be inserted again at its original position (*e.g.*, when the algorithm that uses the list backtracks). Also the pointers $L(x)$ and $R(x)$ remain available. The implementation should guarantee that if a sequence of removals and insertions is performed in a last-out-first-in order, then the linear ordering induced on the elements by the list is preserved.

The mCRL2 model. The list data structure is described by a pair of functions `L` and `R` that for each element gives its left and right neighbours. For simplicity, the elements of the data structure are taken to be natural numbers from 0 up till and including some natural number `MAX`, set to `4` in the specification below.

The functions `remove` and `insert` are described using function updates, and they strictly follow the definition above.

```
sort D = struct pair(L: Nat -> Nat, R: Nat -> Nat);

map MAX: Nat;
eqn MAX = 4;

map remove,insert: Nat # D -> D;
var x: Nat; p: D;
eqn remove(x, p) = pair(L(p)[R(p)(x) -> L(p)(x)], R(p)[L(p)(x) -> R(p)(x)]);
    insert(x, p) = pair(L(p)[R(p)(x) -> x], R(p)[L(p)(x) -> x]);
```

For the formulation in the parameterised modal μ-calculus of the correctness claim mentioned above, we define a process `UseList(d,stack)`, which has a doubly linked list `d` and a `stack` as parameters; the `stack` stores the removed elements and allows them to be reinserted in a last-out-first-in order. Initially, the list is full and the stack is empty:

```
map d_full: D;
eqn d_full = pair(lambda n:Nat.if(n == 0, MAX, max(0, n - 1)),
                  lambda n:Nat.if(n == MAX, 0, n + 1));

init UseList(d_full,[]);
```

The process `UseList(d, stack)` executes actions `do_remove` and `do_insert`, representing the activities of removing and inserting an element in the linked list, respectively. An element can only be removed from the list if it is in the list. Since, in our model, the full list contains the numbers `0` to `MAX`, whether some element `x` is in the list can be determined by checking whether `x` is between `0` and `MAX` and not an element of the `stack`. Only the top of a non-empty `stack`, represented by `head(stack)`, can be re-inserted in the list. The current list structure is exposed by the actions `left(x,L(d)(x))` and `right(x,R(d)(x))`; *e.g.*, if, for some element `x` of the list, the action `left(x,x')` is enabled in some state, then this means that `x'` is `L(x)`.

```
act do_remove, do_insert: Nat;
    left, right: Nat # Nat;

proc UseList(d: D, stack: List(Nat))=
  sum x: Nat. (x > 0 && x <= MAX && !(x in stack)) ->
    do_remove(x).UseList(remove(x, d), x |> stack) +
  (stack != []) ->
    do_insert(head(stack)).UseList(insert(head(stack), d), tail(stack)) +
  sum x: Nat. (x <= MAX && !(x in stack)) ->
    (left(x, L(d)(x)) + right(x, R(d)(x))).UseList(d, stack);
```

The analysis. It needs to be verified that, at all times, the operations of removing and inserting an element indeed have the intended removal and insertion effects and that, moreover, the linear ordering induced by the list structure on the elements currently in the list is still consistent with the ordering induced by the list structure on the initial full list. Note that, by our definition of `d_full`, the ordering induced by the list structure on the initial full list is the standard ordering on the natural numbers between `0` and `MAX`, so in the formula below we can refer to the ordering induced by the list structure on the initial full list by simply referring to the standard ordering on the natural numbers.

```
nu X(s: Set(Nat) = {n: Nat | n <= MAX}).
  (forall x: Nat. [do_insert(x)](val(!(x in s)) && X(s + {x})) &&
                  [do_remove(x)](val(x in s) && X(s - {x}))) &&
  (forall x,x': Nat. val(x in s) =>
    [right(x,x')]( X(s) && val(x' in s) &&
      forall x'': Nat.(
         val(x < x'' && x'' < x')
      || val(x' <= x && x < x'' && x'' <= MAX)
      || val(x' <= x && 0 <= x'' && x'' < x')
      ) => val(!(x'' in s)) )
  ) &&
  (forall x,x': Nat. val(x in s) =>
    [left(x,x')]( X(s) && val(x' in s) &&
      forall x'': Nat.(
         val(x' < x'' && x'' < x)
      || val(x <= x' && x' < x'' && x'' <= MAX)
      || val(x <= x' && 0 < x'' && x'' < x)
      ) => val(!(x'' in s)) )
  )
```

The formula needs to express an invariant that holds for all reachable states, so we use a greatest fixed point. The parameter of the formula is a set `s`, which contains the natural numbers currently in the list. The subformula under the greatest fixed point operator expresses that

- the actions `do_remove(x)` and `do_insert(x)` can only take place when the element `x` can be removed or inserted, respectively, and that their execution results in behaviour that is in accordance with a list structure from which `x` has been removed or to which `x` has been added, respectively;
- whenever an action `right(x,x')` is enabled, then `x'` is indeed the next element in the list: it is either the least natural number larger than `x` in the list, or it is less or equal to `x` and natural numbers larger than `x` or smaller than `x'` are not in the list; an analogous property should hold when an action `left(x,x')` is enabled,

The formula can be verified for reasonably large numbers of `MAX`. It can be investigated whether the last-out-first-in order of removals and insertions is essential by replacing the `stack` parameter of the `UseList` process by a set. Doing so, we find that for values of `MAX` greater than 1 the correctness requirement fails.

6 Concurrent Data Structures

6.1 Treiber's stack

In programming it is often necessary to keep track of shared resource elements, such as chunks of memory, that are free to use. Linked lists are often used to keep track of available resources, either using a first-in-first-out or a last-in-first-out strategy. When processes are using such a list concurrently, the standard sequential release and retrieve operations do not suffice any more. It is tempting to use *compare-and-swap* operations for insertion and deletion of list elements in that case. However, R. Kent Treiber showed in [26] that this does not work either. When using compare-and-swap the so-called *ABA problem* causes a problem, cf. [8]. Treiber showed that a double compare-and-swap operation is required. The ensuing data structure is a last-in-first-out linked-list that is commonly referred to as Treiber's stack.

The erroneous compare-and-swap implementation is interesting because it is very hard to find out by means of testing that the implementation is incorrect, especially when the number of elements in the list grows. Even with a dedicated testing scheme, it may take hundreds of millions of insertions and deletions in the list before the erroneous situation is encountered. However, when the error occurs, the list structure is in total disarray, leading to elements in the list becoming inaccessible and lost for use. Probably even worse, it allows elements to be simultaneously used by different processes. In practice this means that software using the faulty implementation can run well for years, but suddenly exhibit erroneous behaviour due to inexplicably messed up data structures.

The Treiber stack is described using a shared linked-list data structure that contains available shared resources v. Each node in the list contains a pointer $v.next$ to the next element in the list. The head of the list is contained in shared variable hd. A shared resource v can be released to the stack using $release(v)$. Resources can be obtained from the stack using *retrieve*. A pseudocode description of releasing and retrieving an element is the following.

release v :
repeat
  $v.next := hd$;
  $b := comp\_and\_swap(hd, v, v.next)$;
until b;

retrieve:
repeat
  $v := hd$;
  if $v \approx 0$ **return** *nothing*;
  $b := comp\_and\_swap(hd, v, v.next)$;
until b;
return v;

The mCRL2 model. We model a situation where two processes `p1` and `p2` share a Treiber stack. The shared linked-list representing the stack is described as follows. The stack consists of `N` elements, that are modelled by natural numbers. One number `hd` represents the head of the list. Function `next: Nat -> Nat` is such that for list element `v`, `next(v)` is the next element in the list. As the data structure is global, it is modelled using a separate process `treibers_stack`

that maintains hd and next. The operations on the data structure are getting and setting a next value in next using actions set_next and get_next, as well as getting the value of hd and setting it using a compare-and-swap action cmp_swp_hd(id,v1,v2,b). When this action takes place, the variable hd is set to v2 provided hd was equal to v1. The boolean b is true when the value was changed. Otherwise, it is false. Note that id is the identity of the process that performs the compare-and-swap.

```
sort ID = struct p1 | p2;
map N: Nat;
eqn N = 2;

act set_next_r, set_next_s, set_next,
    get_next_r, get_next_s, get_next: ID # Nat # Nat;
    cmpswp_hd_r, cmpswp_hd_s, cmpswp_hd: ID # Nat # Nat # Bool;
    get_head_r, get_head_s, get_head: ID # Nat;
    nothing: ID;
    retrieve, release: ID # Nat;

proc
  treibers_stack(hd: Nat, next: Nat -> Nat) =
    sum id: ID, a,v: Nat. set_next_r(id, a, v).treibers_stack(hd, next[a -> v]) +
    sum id: ID, a: Nat. get_next_s(id, a, next(a)).treibers_stack(hd, next) +
    sum id: ID. get_head_s(id, hd).treibers_stack(hd, next) +
    sum id: ID, v1,v2: Nat. cmpswp_hd_r(id, v1, v2, hd==v1).
      treibers_stack(if(hd==v1, v2, hd), next);
```

Process P(id, owns) with identifier id retrieves resources from the stack and releases resources to it. The resources it currently owns are stored in owns. The process can either retrieve an element, which is then added to owns, except if the list is empty in which case no element is obtained, or it can release one of the elements it owns. These procedures have been encoded in the processes release_elmt and retrieve_elmt. Note that there are two actions release (id, v) and retrieve(id, v) that are used to signal that an element v is released to or retrieved from the list. Action nothing(id) indicates that process id tried to retrieve an element from the empty stack.

```
proc
  release_elmnt(id: ID, v: Nat, owns: Set(Nat)) =
    sum hd: Nat. get_head_r(id, hd).
    set_next_s(id, v, hd).
    sum b: Bool. cmpswp_hd_s(id, hd, v, b).
    (b -> P(id, owns-{v})
       <> release_elmnt(id, v, owns));

  retrieve_elmnt(id: ID, owns: Set(Nat)) =
    sum v: Nat. get_head_r(id, v).
    ((v==0) -> nothing(id).P(id, owns)
            <> (sum v_next: Nat. get_next_r(id, v, v_next).
                sum b: Bool. cmpswp_hd_s(id, v, v_next, b).
                (b -> retrieve(id,v).P(id, owns+{v})
                   <> retrieve_elmnt(id, owns))));

  P(id: ID, owns: Set(Nat)) =
    retrieve_elmnt(id, owns) +
    sum v: Nat. (v in owns) -> release(id, v).release_elmnt(id, v, owns);
```

The data structure is initialised in the init section by setting hd to N, and linking each element 1 to 1-1. The number 0 is used as a *null*-pointer, *i.e.*, an indication for the empty list.

```
init allow({ set_next, get_next, cmpswp_hd, get_head,
             nothing, retrieve, release },
       comm({ set_next_r|set_next_s -> set_next,
              get_next_r|get_next_s -> get_next,
              cmpswp_hd_r|cmpswp_hd_s -> cmpswp_hd,
              get_head_r|get_head_s -> get_head },
         treibers_stack(N, lambda l: Nat. max(0, l-1)) ||
         P(p1, {}) || P(p2, {})));
```

The analysis. There are two major properties of Treiber's stack that we want to hold. The first property essentially is the following. Provided that only elements are released that are not in the list, processes can only retrieve elements that are supposed to be in the list. This is expressed in the following formula, that is structurally similar to the last property for Peterson's algorithm. Parameter `free` represents the set of elements supposed to be in the list, and after `retrieve` and `release` actions, it is updated accordingly. The assumption on `release` is characterised using the implication in **`val`**`(!(n in free))=> X(free + {n})`.

```
nu X(free: Set(Nat) = { n:Nat | 0 < n && n <= N }).
  (forall id: ID, n: Nat.
    [release(id, n)](val(!(n in free)) => X(free + {n})) &&
    [retrieve(id, n)](val(n in free) && X(free - {n}))) &&
  [!exists id: ID, n: Nat. (release(id, n) || retrieve(id, n))]X(free)
```

The second property states that at any moment when there are at least two elements in the list, it is always possible to retrieve an element from the list within a finite number of actions while no elements can be released. The reason that there must be at least two elements in the list is that the other process can already have put a claim on an element in the list, without actually having retrieved it. The first five lines of the property follow a structure similar to properties we have seen before. The condition **`val`**`(exists n: Nat. (n in free && free - {n} != {})` encodes that `free` contains at least two elements (it contains `n`, and after removing it, the set is non-empty). As sets can contain infinitely many elements, there is no function in mCRL2 that yields the size of a set. The last two lines say that along all paths not involving `release` actions, a `retrieve` action must inevitably happen.

```
nu X(free: Set(Nat) = { n:Nat | 0 < n && n <= N }).
  (forall id: ID, n: Nat.
    [release(id, n)]X(free + {n}) &&
    [retrieve(id, n)]X(free - {n})) &&
  [!exists id: ID, n: Nat. (release(id, n) || retrieve(id, n))]X(free) &&
  (val(exists n: Nat. (n in free && free - {n} != {})) =>
    mu Y.([(!exists id: ID, n: Nat. retrieve(id, n)) &&
           (!exists id: ID, n: Nat. release(id, n))]Y &&
      <!exists id: ID, n: Nat. release(id, n)>true))
```

It turns out that both formulas are not valid for Treiber's stack when implemented using compare-and-swap, even not so if there are initially two elements only. In Figure 2 we depict two counter examples by drawing the working of the operations on the list. These are the shortest counterexamples that exist, as we used the modal formula prover in breadth first search mode.

The sequence of pictures at the left of the figure shows that it is possible that element `1` is retrieved twice from the data structure without being returned

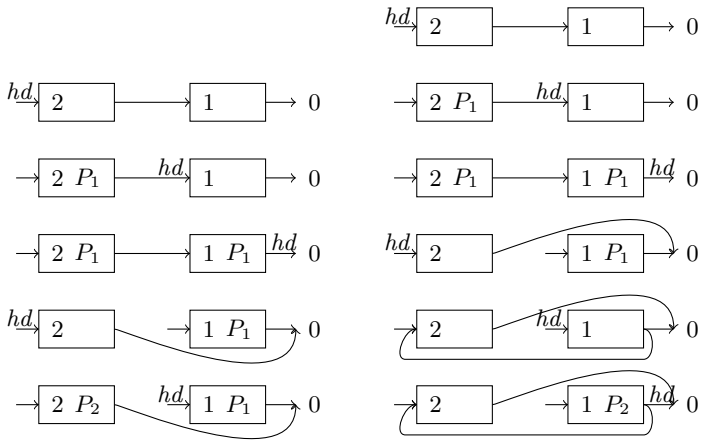

Fig. 2. Counter examples showing that Treiber's stack is incorrect

in between. Initially, process `p2` attempts to retrieve the first data element, but it stops before doing the compare-and-swap. So, `v = 2` and `v_next = 1`. Then process `p1` obtains both element `2` and `1` in a proper fashion. This brings us to the third diagram where `hd` is `0`; the list is empty. Process `p1` returns element `2` in the fourth picture. Now, process `p2` carries out its compare-and-swap, setting `hd` to `1`. The next element that will be released by the stack is `1` despite the fact that is currently owned by process `p1`.

The pictures at the right show how the second property is violated. Process `p1` first obtains element `2`. Subsequently, process `p2` attempts to get element `1` setting `v = 1` and `v_next = 0`, but again the compare-and-swap is not carried out. Process `p1` obtains element `1` and returns subsequently element `2` and then `1`. This yields the fifth picture from above. Process `p2` carries out its compare-and-swap. As `hd = 1` this is successful obtaining the situation at the list at the bottom right. If process `p2` starts to release element `1`, indicated by `release(p2,1)`, but has not yet carried out the compare-and-swap, the variable `free` in the modal formula equals `{1, 2}`, so the free list contains at least two elements. However, `hd = 0`, so, process `p1` will repeatedly fail to get an element from the list. Note that these failures can persist indefinitely, as long as `p1` never completes the compare-and-swap.

Treiber's stack is correct when the compare-and-swap operation in the process `retrieve_elt` is replaced by a double compare-and-swap that checks both *hd* and *hd.next* have the expected value before updating them. Concretely,

```
sum b: Bool. cmpswp_hd_s(id, v, v_next, b).
```

is replaced with

```
sum b: Bool. double_cmpswp_s(id, v, v_next, v, v_next, b).
```

To support the double compare-and-swap operation, the `treibers_stack` process must be extended with the following:

```
sum id: ID, hd_old, hd_new, a, v_old: Nat.
  double_cmpswp_r(id, hd_old, hd_new, a, v_old, hd==hd_old && next(a) == v_old).
  treibers_stack(if(hd==hd_old && next(a)==v_old, hd_new, hd), next);
```

With this change Treiber's stack satisfies both correctness properties, which we could also check for slightly larger stacks.

When verifying the incorrect and correct versions of Treiber's stack one may observe that the state space of the incorrect version is much larger (approximately 300M states for `N=4`) than the correct version. This is a pattern that is more commonly observed in practice, and suggests that, when model checking, one should always start by analysing the smallest conceivable instance of a problems, and only increase instance sizes when these smallest instances satisfy all desired properties.

6.2 Lamport's queue

Bounded single-producer/single-consumer queues have been studied extensively. A classical, and simple description of such a queue is given by Lamport [20]: a queue is represented by an array Q of size N, and is indexed $0 \ldots N-1$. The queue is stored in a circular way, using indices *head* and *tail* to represent the pointers to the head and tail of the queue. Accesses to these pointers are assumed to be atomic. Reading and writing of array elements are non-atomic.

The queue supports two operations, push and pop, that are assumed to be executed from different threads. The producer repeatedly pushes elements to the queue, and the consumer repeatedly pops elements from the queue. When the queue is full, the operation push blocks until an element is removed from the queue. Likewise, when the queue is empty, the operation pop blocks until an element is added to the queue. The algorithm can be described in pseudocode as follows. Note that the algorithm is *wait-free*.

Global variables:	Behaviour push(v) :	Behaviour pop :
$Q[0 \ldots N)$: *Value*	**do**	**do**
head: $\mathbb{N}$	$t := tail$	$t := tail$
tail: $\mathbb{N}$	$h := head$	$h := head$
	while $(t+1) \bmod N = h$	**while** $t = h$
	$Q[t] := v$	$v := Q[h]$
	$tail := (t+1) \bmod N$	$head := (h+1) \bmod N$

Variables t and h in both procedures are local. For correctness, the algorithm assumes so-called sequential consistency [19], *i.e.*, the operations in each of the procedures are executed in the order in which they appear in the program.

The mCRL2 model. The indices in the array are modelled as natural numbers. Values in the queue are represented using finite sort `Value`; the special value `garbage` represents values in the array for which no concrete value is known (either the location is uninitialised, or there is an incomplete write to the location).

```
sort Value = struct garbage | d0 | d1;
```

The head pointer is modelled using the process `Head(i)`. Parameter `i` stores the index to which `Head` points. The parameter can be set to an arbitrary index `i'`, and the process continues recursively with this new index as its parameter. Alternatively, `Head` can return the current index it points to; the value of the parameter is not changed in this case. The tail pointer is modelled analogously. The mCRL2 code is as follows:

```
proc
  Head(i: Nat) = sum i': Nat. set_head_r(i').Head(i') + get_head_s(i).Head();
  Tail(i: Nat) = sum i': Nat. set_tail_r(i').Tail(i') + get_tail_s(i).Tail();
```

The queue of size `N` is modelled using `N` individual instances of the process `Queue(i, v)`. Each of the instances represents a single position in the array, with index `i`, and value `v` that is currently stored at that position. Reads and writes are considered to be non-atomic. Therefore, writes are modelled using `start_write_queue` and `end_write_queue`. Reads are modelled using `start_read_queue` and `end_write_queue`. To accurately model non-atomicity of these operations, we keep track of threads currently reading from the position, writing to it (we store the value being written), or both. When a thread is writing, we store the value `garbage` in `v`. This enables us to verify that no value is ever read while another thread is writing.

```
map N:Pos;
eqn N=2;
proc
  Queue(i: Nat, v: Value) =
    sum v': Value. start_write_queue_r(i, v').QueueW(v = garbage, w = v') +
    start_read_queue_s(i).QueueR();
  QueueW(i: Nat, v: Value, w: Value) =
    end_write_queue_r(i).Queue(v = w) +
    start_read_queue_s(i).QueueRW();
  QueueR(i: Nat, v: Value) =
    sum v': Value. start_write_queue_r(i, v').QueueRW(v = garbage, w = v') +
    end_read_queue_s(i, v).Queue();
  QueueRW(i: Nat, v: Value, w: Value) =
    end_write_queue_r(i).QueueR(v = w) +
    end_read_queue_s(i, v).QueueW();
```

The `Producer` repeatedly `Push`-es an arbitrary (non-garbage) value to the queue. The `Push` process first gets the values of the tail and head pointer from the respective variables and keeps track of them locally in `t` and `h`. If the queue is full, the process blocks: it will loop and get the head and tail pointers again until space becomes available. Note that this is a less abstract way of modelling busy-waiting than that which was chosen in Peterson's algorithm. If the queue is not full, it will non-atomically store value `v` to the tail position `t` that was just obtained, and (atomically) update the tail pointer to `(t+1)mod n`. The `Pop` process is modelled in a similar way.

```
proc
  Producer = sum v: Value. (v != garbage) -> call_push(v).Push(v).Producer;
  Push(v:Value) =
    sum t: Nat. get_tail_r(t).sum h: Nat. get_head_r(h).
    (((t+1) mod N == h) -> Push()
        <> ( start_write_queue_s(t, v).end_write_queue_s(t).
             set_tail_s((t+1) mod N).
```

```
            ret_push ));

  Consumer = call_pop.Pop.Consumer;
  Pop =
    sum t: Nat. get_tail_r(t).sum h: Nat. get_head_r(h).
    ((t == h) -> Pop
        <> ( start_read_queue_r(h).
             sum v: Value. end_read_queue_r(h, v).
             set_head_s((h+1) mod N).
             ret_pop(v) ));
```

The process is initialised as follows:

```
init
  allow({ start_read_queue, end_read_queue, start_write_queue,
          end_write_queue, get_head, set_head, get_tail, set_tail,
          call_push, ret_push, call_pop, ret_pop },
    comm({ start_read_queue_s | start_read_queue_r -> start_read_queue,
           end_read_queue_s | end_read_queue_r -> end_read_queue,
           start_write_queue_r | start_write_queue_s -> start_write_queue,
           end_write_queue_r | end_write_queue_s -> end_write_queue,
           get_head_s | get_head_r -> get_head,
           set_head_r | set_head_s -> set_head,
           get_tail_s | get_tail_r -> get_tail,
           set_tail_r | set_tail_s -> set_tail },
      Queue(0,garbage) || Queue(1,garbage) || Head(0) || Tail(0) ||
      Producer || Consumer));
```

The analysis. The main property we want to verify for Lamport's queue is that it actually behaves as a queue. The queue is *first-in-first-out*, and the capacity is never exceeded. Together, this is expressed in the following μ-calculus formula.

```
nu X(q: List(Value) = []).
  [ret_push](val(#q <= N)) &&
  forall v: Value.
    [call_push(v)](val(v != garbage) && X(v |> q)) &&
    [ret_pop(v)](val(q != [] && v == rhead(q)) && X(rtail(q))) &&
  [!(exists v': Value. call_push(v') || ret_pop(v'))] X(q)
```

This first order μ-calculus formula, similar to what we have seen before, keeps track of the contents of the queue in parameter `q`, which is a list of values. After every completion of a push operation it checks the current size of the queue is at most `N`. For every call to `push` with value `v`, it is verified that the value that is pushed is not `garbage`, and it recursively verifies the property for the queue extended with value `v`. Likewise, for every value that is returned by `pop`, it is verified that the queue was not empty, and the value that is returned corresponds to the oldest value in the queue. For all other values, `q` is checked recursively again. Since we use the greatest fixed point, we verify an invariant.

We can also verify other properties for the queue. For instance, every call to `push` is guaranteed to terminate. This is expressed as follows.

```
[true*.exists v: Value. call_push(v)]( mu X.[!ret_push]X && <true>true )
```

This property does not hold for the queue. A counterexample is the infinite sequence in which subsequence `get_tail(0).get_head(0)` is repeated indefinitely. In this case, the consumer always checks whether an element is available in the queue, but the producer never produces any value. Instead, the following property holds. It expresses that after `call_push`, it remains possible to do a `ret_push` as long as it has not been done yet.

```
[true*.exists v: Value. call_push(v).!ret_push*]<!ret_push* . ret_push>true
```

We next verify some properties for the `head` pointer; similar properties hold for `tail`. First, whenever the pointer to head is set, subsequent reads are guaranteed to return the same value, until the pointer is set again.

```
[true*] forall i: Nat. [set_head(i).(!exists j: Nat.set_head(j))*]
    forall i': Nat.[get_head(i')] val(i==i')
```

Next, the pointer is never set out of bounds.

```
[true*] forall i :Nat. [set_head(i)] val(i<N)
```

Finally, we verify that the `push` function never tries to write out of bounds. Due to the way synchronisation works, the calls `start_set_queue(i,v)` and `start_get_queue(i)` with $i \geqslant N$ will not be visible in the process as we modelled it so far. So, attempts to access these positions will silently fail. This can be circumvented by adding a process `Invalid` into the parallel composition with the following specification.

```
Invalid =
  sum i: Nat, v: Value. (i>=N) -> start_write_queue_r(i,v).Invalid +
  sum i: Nat. (i>=N) -> start_read_queue_s(i).Invalid;
```

If we incorporate this process, we can verify the absence of out of bounds writes using the following invariant.

```
[true*.exists i: Nat, v: Value. val(i>=N) && start_set_queue(i, v)] false
```

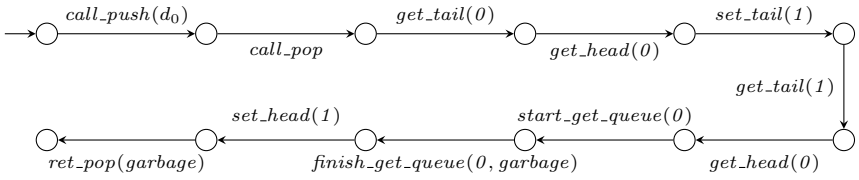

Fig. 3. Counterexample for Lamport's queue with no sequential consistency

As mentioned before, Lamport's queue assumes sequential consistency. If we drop this requirement, since there is no dependency between the assignment to $Q[t]$ and the assignment to *tail* in the *push* routine, the compiler may reorder these statements. If we change the mCRL2 model accordingly, and we verify whether the process behaves as a queue according to the first property stated above, the tool will observe the property does not hold. The counterexample that it generates is shown in Figure 3. From this example we clearly see that the consumer reads invalid values from the queue. The underlying reason for this is that variable *head* is used for synchronisation between the producer and consumer; incrementing this variable signals to the consumer that a new value has been added to the queue. However, when allowing for reordering of operations, this is now signalled before the write has completed.

References

1. https://github.com/mCRL2org/mCRL2/tree/master/examples/software_models.
2. M. Bartholomeus, B. Luttik, and T. Willemse. Modelling and Analysing ERTMS Hybrid Level 3 with the mCRL2 Toolset. In *Formal Methods for Industrial Critical Systems*, pages 98–114. Springer, Cham, September 2018.
3. Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2004.
4. R. van Beusekom, J.F. Groote, P.F. Hoogendijk, R. Howe, W. Wesselink, R. Wieringa, and T.A.C. Willemse. Formalising the dezyne modelling language in mCRL2. In *FMICS-AVoCS*, volume 10471 of *Lecture Notes in Computer Science*, pages 217–233. Springer, 2017.
5. M. Bouwman, B. Janssen, and B. Luttik. Formal modelling and verification of an interlocking using mCRL2. In *FMICS*, volume 11687 of *Lecture Notes in Computer Science*, pages 22–39. Springer, 2019.
6. O. Bunte, J.F. Groote, J.J.A. Keiren, M. Laveaux, T. Neele, E.P. de Vink, W. Wesselink, A. Wijs, and T.A.C. Willemse. The mCRL2 toolset for analysing concurrent systems. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 21–39. Springer, Cham, April 2019.
7. A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV 2: An opensource tool for symbolic model checking. In E. Brinksma and K.G. Larsen, editors, *Computer Aided Verification*, Lecture Notes in Computer Science, pages 359–364, Berlin, Heidelberg, 2002. Springer.
8. D. Dechev, P. Pirkelbauer, and B. Stroustrup. Understanding and effectively preventing the ABA problem in descriptor-based lock-free designs. In *13th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing, ISORC 2010, Carmona, Sevilla, Spain, 5-6 May 2010*, pages 185–192. IEEE Computer Society, 2010.
9. G. Ernst, M. Huisman, W. Mostowski, and M. Ulbrich. VerifyThis - verification competition with a human factor. In D. Beyer, M. Huisman, F. Kordon, and B. Steffen, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 25 Years of TACAS: TOOLympics, Held as Part of ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings, Part III*, volume 11429 of *Lecture Notes in Computer Science*, pages 176–195. Springer, 2019.
10. H. Garavel, F. Lang, R. Mateescu, and W. Serwe. CADP 2011: A Toolbox for the Construction and Analysis of Distributed Processes. *STTT*, 15(2):89–107, 2013.
11. T. Gibson-Robinson, P. Armstrong, A. Boulgakov, and A.W. Roscoe. FDR3 — A modern refinement checker for CSP. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 187–201. Springer, Berlin, Heidelberg, April 2014.
12. J.F. Groote and M.R. Mousavi. *Modeling and Analysis of Communicating Systems*. MIT Press, 2014.
13. M. Hennessy and R. Milner. Algebraic laws for nondeterminism and concurrency. *J. ACM*, 32(1):137–161, 1985.
14. H. Hitotumatu and K. Noshita. A technique for implementing backtrack algorithms and its application. *Inf. Process. Lett.*, 8(4):174–175, 1979.
15. G.J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.

16. Y.L. Hwong, J.J.A. Keiren, V.J.J. Kusters, S. Leemans, and T.A.C. Willemse. Formalising and analysing the control software of the Compact Muon Solenoid Experiment at the Large Hadron Collider. *Science of Computer Programming*, 78(12):2435–2452, December 2013.
17. J.J.A. Keiren and M.D. Klabbers. Modelling and verifying IEEE Std 11073-20601 session setup using mCRL2. *Electronic Communications of the EASST*, 53(0), January 2013.
18. D.E. Knuth. Dancing links, 2000. arXiv cs/0011047.
19. L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690–691, 1979.
20. L. Lamport. Specifying concurrent program modules. *ACM Transactions on Programming Languages and Systems*, 5(2):190–222, 1983.
21. M. Laveaux, J.F. Groote, and T.A.C. Willemse. Correct and efficient antichain algorithms for refinement checking. In *FORTE*, volume 11535 of *Lecture Notes in Computer Science*, pages 185–203. Springer, 2019.
22. T. Nipkow, L.C. Paulson, and M. Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002.
23. G.L. Peterson. Myths about the mutual exclusion problem. *Information Processing Letters*, 12(3):115—-116, 1981.
24. D. Remenska, T.A.C. Willemse, K. Verstoep, J. Templon, and H. Bal. Using model checking to analyze the system behavior of the LHC production grid. *Future Generation Computer Systems*, 29(8):2239–2251, October 2013.
25. A.W. Roscoe. *Understanding Concurrent Systems*. Texts in Computer Science. Springer London, London, 2010.
26. R.K. Treiber. Systems programming: Coping with parallelism. Technical Report RJ 5118 (53162), International Business Machines Incorporated, Thomas J. Watson Research Center, San Jose, California, 1986.
27. Wikipedia. http://en.wikipedia.org/wiki/peterson's_algorithm, 2015. Retrieved May 17, 2015.

Science Reports **Department of Mathematics and Computer Science**
Technische Universiteit Eindhoven

If you want to receive reports, send an email to: a.gouma@tue.nl (we cannot guarantee the availability of the requested reports).

In this series appeared (from 2015):

15/01	Önder Babur, Tom Verhoeff and Mark van den Brand	Multiphysics and Multiscale Software Frameworks: An Annotated Bibliography
15/02	Various	Proceedings of the First International Workshop on Investigating Dataflow In Embedded computing Architectures (IDEA 2015)
15/03	Hrishikesh Salunkhe, Alok Lele, Orlando Moreira and Kees van Berkel	Buffer Allocation for Realtime Streaming Applications Running on a Multi-processor without Back-pressure
15/04	J.G.M. Mengerink, R.R.H. Schiffelers, A. Serebrenik, M.G.J. van den Brand	Evolution Specification Evaluation in Industrial MDSE Ecosystems
15/05	Sarmen Keshishzadeh and Jan Friso Groote	Exact Real Arithmetic with Pertubation Analysis and Proof of Correctness
15/06	Jan Friso Groote and Anton Wijs	An $O(m \log n)$ Algorithm for Stuttering Equivalence and Branching Bisimulation
17/01	Ammar Osaiweran, Jelena Marincic Jan Friso Groote	Assessing the quality of tabular state machines through metrics
17/02	J.F. Groote and e.P. de Vink	Problem solving using process algebra considered insightful
18/01	L. Sanchez, W. Wesselink and T.A.C. Willemse	BDD-Based Parity Game Solving: A comparison of Zielonka's Recursive Algorithm, Priority Promotion and Fixpoint Iteration
18/02	Mahmoud Talebi	First-order Closure Approximations for Middle-Sized Systems with Non-linear Rates
18/03	Thomas Neele, Tim A.C. Willemse and Jan Friso Groote	Solving Parameterised Boolean Equation Systems with Infinite Data Through Quotienting (Technical Report)
18/04	Daan Leermakers, Behrooz Razeghi, Shideh Rezaeifar, Boris Škorić, Olga Taran Slava Voloshynovskiy	Optical PUF statistics
19/01	Maurice Laveaux, Jan Friso Groote and Tim A.C. Willemse	Correct and Efficient Antichain Algorithms for Refinement Checking
19/02	Thomas Neel, Tim A.C. Willemse and Wieger Wesselink	Partial-Order Reduction for Parity Games with an Application on Parameterised Boolean Equation Systems (Technical Report)
19/03	David N. Jansen, Jan Friso Groote, Jeroen J.A. Keiren and Anton Wijs	A simpler $O(m \log n)$ algorithm for branching bisimilarity on labelled transition systems
19/04	Alexander Fedotov, Jeroen J.A. Keiren and Julien Schmaltz	Sound idle and block equations for finite state machines in xMAS
19/05	J.F. Groote, J.J.A. Keiren, B. Luttik, E.P. de Vink and T.A.C. Willemse	Modelling and Analysing Software in mCRL2