Model Checking and Performance Evaluation with CADP Illustrated on Shared-Memory Mutual Exclusion Protocols

Radu Mateescu, Wendelin Serwe

Model Checking and Performance Evaluation with CADP Illustrated on Shared-Memory Mutual Exclusion Protocols

Radu Mateescu[a], Wendelin Serwe[a,*]

[a]INRIA *Grenoble – Rhône-Alpes* / VASY / LIG
Inovallée, 655, av. de l'Europe, F-38334 Saint Ismier, France

Abstract

Mutual exclusion protocols are an essential building block of concurrent shared-memory systems: indeed, such a protocol is required whenever a shared resource has to be protected against concurrent non-atomic accesses. Hence, many variants of mutual exclusion protocols exist, such as Peterson's or Dekker's well-known protocols. Although the functional correctness of these protocols has been studied extensively, relatively little attention has been paid to their non-functional aspects, such as their performance in the long run. In this paper, we report on experiments with the CADP toolbox for model checking and performance evaluation of mutual exclusion protocols using Interactive Markov Chains. Steady-state analysis provides an additional criterion for comparing protocols, which complements the verification of their functional properties. We also carefully re-examined the functional properties of these protocols, whose accurate formulation as temporal logic formulas in the action-based setting turns out to be quite involved.

Keywords: functional verification, interactive markov chain, LOTOS NT, μ-calculus, process algebra, steady-state simulation

1. Introduction

Mutual exclusion is a long-standing problem in concurrent programming, formulated initially by Dijkstra almost half a century ago [1]. It consists in controlling the access of concurrent processes to a shared resource such that at most one process can use the resource at a time and that the execution of the protocol is guaranteed not to prevent the system from progressing. In the shared-memory setting, in which processes communicate by atomic read and write operations on shared variables, a large number of protocols implementing mutual exclusion were proposed and studied in the literature (see, e.g., the surveys in [2–4]). Most of the effort has been concentrated on analyzing the functional correctness of these protocols, either by hand-written proofs [1, 3, 5–10] or by applying automated reasoning and model checking techniques [11–14]. However, much less attention has been given to the model-based performance evaluation of these

*Corresponding author
Email addresses: `radu.mateescu@inria.fr` (Radu Mateescu), `wendelin.serwe@inria.fr` (Wendelin Serwe)

protocols, most of the existing works dealing with performance measurements of protocol implementations on specific architectures [15, 16, 32]. Model-based approaches have the advantage that they enable the analysis of systems that are yet to be built, and to guide designers in their choices [24].

In this paper, we show how Interactive Markov Chains (IMC) [17] and their implementation in the CADP verification toolbox [18] can be applied to the model checking and performance analysis of shared-memory mutual exclusion protocols. We assume that only the mean values of actual durations are known, which can be modeled conveniently using exponentially distributed durations in the IMC setting. If more concrete duration information is available, this can be encoded using IMCs by means of phase-type distributions [19], which can be employed as precise approximations of arbitrary (discrete or continuous) probability distributions.

As the high-level specification language for IMCs, we use LOTOS NT [20, 21], a process-algebraic language with imperative flavor accepted as input by CADP. We study the stochastic behavior of these protocols in the long run by further transforming the IMCs generated from LOTOS NT specifications into continuous-time Markov chains (in which nondeterminism is solved by a uniform scheduler) and analyzing them using the BCG_STEADY [22] and CUNCTATOR [23] tools of CADP, which computes (respectively, approximates) the throughputs of various actions at steady-state. This allows us to compare the performance of various protocols and to study the impact of certain parameters (e.g., number of processes, caching, relative speed of processes, fraction of time taken by critical sections, etc.) on the performance of the system and/or of individual processes. Another useful measure that can be obtained from steady-state analysis is the mean number of accesses to shared variables performed by each process [24].

One advantage of IMCs is that the *same* specification of a protocol can be used for both performance evaluation and functional verification [25]. Although mutual exclusion protocols serve traditionally as basic examples to illustrate the use of model checkers, it is not obvious how to find an accurate description of their correctness properties in the action-based setting, in particular, for more than two processes. We revisit these properties and specify them concisely using MCL, an extension of alternation-free modal μ-calculus with data-handling constructs and fairness operators accepted as input by the EVALUATOR 4.0 [26] on-the-fly model checker. We observe that certain important properties are linear-time in nature, requiring formulas of $L\mu_2$ (the μ-calculus of alternation depth two) [27] or ACTL* [28]. Using MCL formulas parameterized by data values, we also apply model checking to determine some non-functional parameters of the protocols, such as the degree of overtaking between processes. The results of model checking (e.g., about the starvation of certain processes) are corroborated by the results of performance evaluation.

This paper extends the study reported in [29] by formally specifying protocols with more than two processes, by generalizing the correctness properties to an arbitrary number of processes, and by investigating the effect of non-uniform memory accesses (notably, caches) on performance.

The paper is organized as follows: Section 2 defines the terminology, shows the encoding of mutual exclusion protocols using LOTOS NT and how the stochastic aspects are incorporated to yield IMC models; Section 3 presents the analysis of the protocols by means of model checking and performance evaluation using CADP; and finally, Section 4 gives some concluding remarks and directions for future work.

2. Background

After a brief reminder of the mutual exclusion problem in the shared-memory setting, we present in this section the modeling of the behavioral and stochastic aspects of mutual exclusion protocols using LOTOS NT.

2.1. Shared-Memory Mutual Exclusion Protocols

We briefly present here the mutual exclusion problem in the shared-memory setting as formulated in [3]. Concurrent processes communicate and synchronize only by means of atomic operations on shared variables. Each process consists of four parts of code, executed cyclically in the following order: non-critical section, entry section, critical section, and exit section. The shared resource can be accessed only in the critical section, and the shared variables can be accessed only in the entry and exit sections. Processes are allowed to stop in their non-critical section but must leave their critical section in a finite amount of time. The entry and exit sections must manipulate the shared variables in such a way that at most one process at a time is in its critical section and the execution of processes is guaranteed to progress (see Sec. 3.1 for a more precise formulation of these properties).

The earliest mutual exclusion protocols allowed only atomic read and write operations on the shared variables[1]. More recent protocols take advantage of more powerful operations provided by modern processors (such as atomic *compare-and-swap*, *fetch-and-store*, or *read-and-increment*), and are designed for cache-coherent multiprocessor architectures. Some protocols use the numerical identifiers of processes to arbitrate between processes trying to access the critical section at the same time, thus giving priority to the process with the smaller or greater number. Finally, although most shared-memory mutual exclusion protocols are designed for an arbitrary number of processes, some protocols are optimized for the case of two processes only (general schemes for their extension to more processes exist [30]).

2.2. Modeling Mutual Exclusion Protocols using LOTOS NT

We specified the mutual exclusion protocols formally using LOTOS NT [20, 21], a variant of the E-LOTOS [31] standard implemented within CADP. LOTOS NT tries to combine the best of process-algebraic languages and imperative programming languages: a user-friendly syntax, common to data types and processes; constructed type definitions and pattern-matching; and imperative statements (assignments, conditionals, loops, etc.). LOTOS NT is supported by the LNT.OPEN tool, which translates LOTOS NT specifications into labeled transition systems (LTSs) suitable for on-the-fly verification using CADP.

Figure 1 shows the LOTOS NT specification of the protocol proposed by Burns & Lynch [6], instantiated for two processes. This protocol uses two shared bits, which we represent as the cells $A[0]$ and $A[1]$ of a two-bit array, in the same way as [30]. The original pseudo-code of the protocol (see Fig. 1(a)) contains conditional jump statements, which are translated in LOTOS NT using "**break**" statements (see Fig. 1(b)). The non-critical

[1]The black-white bakery protocol [10] even allows a write operation to be simultaneous with several read operations.

```
  loop
     non-critical section;
$L_0$: A[i] := 0;
     if i = 1 and A[0] = 1 then
       goto $L_0$
     end if;
     A[i] := 1;
     if i = 1 and A[0] = 1 then
       goto $L_0$
     end if;
$L_1$: if i = 0 and A[1] = 1 then
       goto $L_1$
     end if;
     critical section;
     A[i] := 0
  end loop
```

(a)

```
process P [NCS:Pid,CS:Access,A:Operation] (i:Nat) is
  loop var a0, a1:Nat in
    NCS (i);
    loop L in
      A (Write, i, 0, i);
      A (Read, 0, ?a0, i);
      if (i == 0) or (a0 == 0) then
        A (Write, i, 1, i);
        A (Read, 0, ?a0, i);
        if (i == 0) or (a0 == 0) then
          break L
        end if
      end if
    end loop;
    A (Read, 1, ?a1, i);
    while (i == 0) and (a1 == 1) loop
      A (Read, 1, ?a1, i)
    end loop;
    CS (Enter, i); CS (Leave, i);
    A (Write, i, 0, i)
  end var end loop
end process
```

(b)

```
par A, CS, NCS in
  par A in
    par
      P [NCS, CS, A] (0)
      ||
      P [NCS, CS, A] (1)
    end par
    ||
    par
      Var [A] (0,0) || Var [A] (1,0)
    end par
  end par
  ||
  L [A, CS, NCS, MU]
end par
```

(c)

```
process Var [A:Operation] (ind, val:Nat) is
  loop
    select
      A (Read, ind, val, ?any Nat)
      []
      A (Write, ind, ?val, ?any Nat)
    end select
  end loop
end process
```

(d)

Figure 1: Burns & Lynch protocol [6] for two processes: (a) Unstructured pseudo-code of process P_i ($i \in \{0, 1\}$); (b) Lotos NT code of process P_i; (c) Lotos NT code of the systems' architecture; (d) Lotos NT code of the cell $A[ind]$ of the shared array.

Protocol	Size ($n = 2$)	
	states	transitions
Anderson	68	136
Burns & Lynch	65	130
B&W Bakery	1818	3636
Clh	246	492
Dekker	144	288
Dijkstra	256	512
Kessels	276	552
Knuth	168	336
Lamport	284	568
Mcs	75	150
Peterson	59	118
Peterson$_t$	65	130
Szymanski	98	196
tas	16	32
ttas	27	54
trivial	20	40

Generated protocols	Size ($n = 2$)	
	states	transitions
2b_p1	63	126
2b_p2	76	152
2b_p3	84	168
3b_p1	88	176
3b_p2	151	302
3b_c_p1_orig	90	180
3b_c_p1	87	174
3b_c_p2	69	138
3b_c_p3	63	126
4b_p1	84	168
4b_p2	188	376
4b_c_p1	227	454
4b_c_p2	227	454

Table 1: State space sizes of the protocols for two processes (minimized for strong bisimulation)

and critical sections are modeled using the (non-synchronized) actions NCS and CS. The operations on a shared variable are modeled as rendezvous synchronizations on gate A with a process Var, which models a cell of the two-bit array (see Fig. 1(d)). Note that process Var is parameterized by a natural number instead of merely a boolean value; this will allow Var to be reused for other protocols involving shared natural numbers.

Emission and reception of values on a gate can take place simultaneously, as in the action "A (Read, 0, ?a_0, i)", where the values Read, 0, and i are emitted and a value is received in variable a_0, previously declared using a "**var**" statement. Gates are typed in Lotos NT: in process P, the types Pid, Access, and Operation denote the communication profiles (i.e., number and types of the exchanged values) of gates NCS, CS, and A, respectively. Note that in the Lts generated by Lnt.Open, a synchronization on gate G involving the communication of values v_1, ..., v_n is represented by an action of the form $G\ v_1\ ...\ v_n$. To facilitate the specification of temporal properties (see Sec. 3.1), the critical section of process i is represented by two actions ("CS (Enter, i)" and "CS (Leave, i)") and each operation on a shared variable carries the identifier of the executing process.

The Lotos NT specification of process P_i follows very closely the pseudo-code of the protocol, but makes explicit all read operations on shared variables before each evaluation of an expression containing these variables. The architecture of the system (see Fig. 1(c)) shows the interconnection of processes and shared variables. For all protocols considered, all shared variables are initialized to 0. We specified 24 mutual exclusion protocols in Lotos NT following the scheme shown in Figure 1: Anderson's array-based queuelock [32], Burns & Lynch [6], Craig and Landin & Hagersten (Clh) [33, 34], Dekker [35], Dijkstra [1], Peterson [7], Knuth [5], Lamport [8], Kessels [36], Mellor-Crummey & Scott (Mcs) [37], Szymanski [9], the black-white bakery protocol [10], and twelve protocols generated automatically in [30]. Additionally, we also specified a trivial (incorrect) one-

Protocol	Size ($n = 3$)		Size ($n = 4$)		Size ($n = 5$)	
	states	transitions	states	transitions	states	transitions
Anderson	474	1,422	3,104	12,416	21,160	105,800
Burns & Lynch	841	4,782	12,950	51,800	235,277	1,176,385
B&W Bakery	174,932	524,796	21,463,998	85,855,992	out of memory	
Clh	5,544	16,632	159,480	637,920	5,834,160	29,170,800
Dijkstra	8,388	25,164	273,247	1,092,988	9,433,726	47,168,630
Knuth	3,174	9,522	67,486	269,944	1,634,490	8,172,450
Lamport	9,939	29,817	278,508	1,114,032	7,993,494	39,967,470
Mcs	810	2,430	10,341	41,364	157,788	788,940
Peterson	2,511	7,533	241,426	965,704	38,105,669	190,528,345
Peterson$_t$	965	2,895	14,025	56,100	394,600	1,973,000
Szymanski	1,971	5,913	50,228	200,912	1,554,216	7,771,080
tas	44	132	112	448	272	1,360
ttas	108	324	405	1,620	1,458	7,290
trivial	60	180	160	640	400	2,000

Table 2: State space sizes of the protocols for more than two processes (minimized for strong bisimulation)

bit protocol for benchmarking purposes and two one-bit semaphore protocols, namely a *test-and-set* (written tas) and *test–test-and-set* (written ttas) [32]. The total size of the specifications (including comments, and after factoring common datatypes and processes in separate modules as much as possible) is about 5300 lines of Lotos NT.

Tables 1 and 2 give the state space sizes of the protocols (minimized for strong bisimulation), for up to five processes, when applicable. Peterson's protocol exists in two versions: the one for n processes presented in [7] (written without index) and a generalization of the optimized protocol for two processes, using a binary tree of binary locks (written with index t). The protocols of [30] were generated for two processes only.

2.3. Transformation to Interactive Markov Chains

Each protocol is transformed into an Interactive Markov Chain (Imc) by adding Markov delays in a constraint-oriented style [25]. Specifically, we compose the Lts corresponding to the protocol in parallel with an Imc associating a delay to each operation of the protocol. This Imc can be described by a Lotos NT process L; Figure 2 shows the process required for Burns & Lynch's protocol.

Because process L is synchronized on all actions A, CS, and NCS, L enforces that each of these actions is followed by a MU action, which can be renamed into a stochastic transition once the Lts corresponding to the Lotos NT specification has been generated. Due to this modeling style, in each Imc generated for the protocols, one can distinguish two kinds of states: on the one hand, the states where process L lets elapse a delay, and on the other hand, the states where each process can execute some interactive action. The former kind of states has fanout 1, because there is only one delay to let elapse. The latter kind of states has fanout n, because each process can execute some (but only one) action at any time.

To model non-uniform memory access times due to local caches of the processors, process L stores, for each shared variable, the status of all caches in a variable of type

```
process L [NCS:Pid, CS:Access, A:Operation, MU:Latency] is
  var
    index, pid: Nat, $a_0$, $a_1$: Cache, sig: Signal, csig: Cache_Signal
  in
    $a_0$ := cache (Invalid); $a_1$ := cache (Invalid);
    loop
      select
        A (?sig, 0 of Nat, ?any Nat, ?pid);
        eval csig := update_caches (pid, sig, !?$a_0$);
        MU (csig, pid)
      []
        A (?sig, 1 of Nat, ?any Nat, ?pid);
        eval csig := update_caches (pid, sig, !?$a_1$);
        MU (csig, pid)
      []
        CS (?sig, ?pid);
        if sig == Enter then
          MU (sig, pid)
        end if
      []
        NCS (?pid);
        MU (Work, pid)
      end select
    end loop
  end var
end process
```

Figure 2: Auxiliary process for inserting Markov delays

"Cache" (variables a_0 and a_1 in Figure 2). Procedure "update_caches" encapsulates the cache coherence protocol (in our case, the MESI protocol [38]) with a write-back policy, which encodes the rules governing changes between the states of the cache lines (Modified, Exclusive, Shared, Invalid). For an operation "sig" (e.g., read) executed by process "pid", the instruction "**eval** csig := update_caches (pid, sig, !?c)" assigns to variable "csig" the corresponding operation taking into account the cache status c (e.g., read_from_local_cache, read_from_remote_cache, or read_from_memory), and also updates the status of the caches (stored in the array c). This approach of handling caching is similar to the approach of [24].

The parameters of action MU enable us to distinguish, for each process, between a read access, a write access (in the local cache or in global memory), a stay in the critical section, and a stay in the non-critical section. We exploit these parameters to experiment with different rates for all of these actions.

Unfortunately, although each process taken separately is deterministic and never blocks (but rather enters a busy-wait loop), the IMCs obtained contain nondeterministic choices whenever several concurrent accesses to shared variables are possible in the same state. To resolve this nondeterminism, we assume the presence of a uniform scheduler, which chooses equiprobably one of the actions (see Sec. 3.2 for technical details). This assumption is based on the fact that a uniform scheduler provides the best choice (in the

Protocol	Size ($n = 2$)	
	states	transitions
Anderson	596	826
Burns & Lynch	465	647
B&W Bakery	22,374	30,802
CLH	3,401	4,708
Dekker	1,156	1,612
Dijkstra	2,682	3,676
Kessels	3,299	4,517
Knuth	1,575	2,159
Lamport	2,424	3,310
MCS	1,269	1,744
Peterson	846	1,178
Peterson$_t$	792	1,102
Szymanski	701	966
tas	88	122
ttas	165	232
trivial	92	127

Generated protocols	Size ($n = 2$)	
	states	transitions
2b_p1	391	541
2b_p2	459	639
2b_p3	553	766
3b_p1	872	1,203
3b_p2	1,572	2,152
3b_c_p1_orig	1,077	1,494
3b_c_p1	1,110	1,540
3b_c_p2	850	1,178
3b_c_p3	894	1,242
4b_p1	2,199	3,029
4b_p2	3,003	4,137
4b_c_p1	2,235	3,053
4b_c_p2	2,235	3,053

Table 3: IMC sizes of the protocols for two processes (using compositional generation)

sense of maximizing entropy [39]) when no additional information is available about the choice of actions performed by the physical system; otherwise, the additional information could be used to specify a particular scheduler, e.g., as an additional LOTOS NT process, leading to different performance analysis results. A more general solution, inspired by a technique used in the context of Markov decision processes [40], would be to consider all possible schedulers to identify the interval (minimum and maximum) of possible throughput values at steady state (an effective procedure for this analysis in the IMC setting was proposed very recently [41], but is not yet available as an implementation).

Practically, the IMC for each protocol is generated compositionally as follows. Because the LTS corresponding to process L alone may be significantly larger than the final IMC (for instance, the LTS of L for the black-white bakery protocol with two processes has $396,577$ states and $84,651,704$ transitions), the LTS of the protocol (see Tables 1 and 2) first serves as interface to generate, using semi-composition [42], a reduced LTS for L. Composing the latter in parallel with the protocol then yields the IMC. The compositional generation has the advantage of yielding a smaller IMC, because the protocol can be minimized before inserting the delays. However, the reduction in IMC size comes at the price of a longer overall generation time, mostly due to large interfaces used in the semi-composition.

Tables 3 and 4 give the state space sizes of the IMCs obtained by compositionally adding delays into the protocols (see Tables 1 and 2), for up to five processes.[2]

The generation of the protocols and IMCs has been automated by an SVL script (30

[2]The IMC sizes of Table 3 differ slightly from those given in [29, Table 1] because the generalization to an arbitrary number of processes required a reformulation of the protocols (which might trigger different optimizations in the LOTOS NT and LOTOS compilers), and because the protocols are now generated compositionally and minimized for strong bisimulation.

Protocol	Size ($n = 3$)		Size ($n = 4$)		Size ($n = 5$)	
	states	transitions	states	transitions	states	transitions
Anderson	25,135	40,941	1,393,476	2,547,912	106,287,227	213,829,735
Burns & Lynch	13,680	21,832	452,729	774,353	16,982,847	30,225,807
B&W Bakery	6,101,735	9,458,933	1,642,516,115	2,709,189,923	out of memory	
Clh	216,540	346,662	15,283,637	26,515,076	1,314,399,178	2,393,291,510
Dijkstra	992,233	1,561,687	176,592,305	300,610,628	out of memory	
Knuth	80,044	124,932	3,909,396	6,528,075	201,320,700	350,053,036
Lamport	234,911	370,523	16,123,731	27,675,696	1,050,516,119	1,906,123,007
Mcs	50,961	80,865	1,794,617	3,085,592	61,003,493	110,128,765
Peterson	171,464	268,874	55,144,244	91,957,592	out of memory	
Peterson$_t$	62,786	100,138	4,814,241	8,213,400	591,372,824	1,078,804,780
Szymanski	28,223	43,735	1,421,916	2,335,428	85,643,362	145,669,082
tas	380	630	1,368	2,556	4,422	9,010
ttas	1,011	1,707	5,193	9,912	24,013	50,165
trivial	390	614	1,325	2,246	3,984	7,080

Table 4: Imc sizes of the protocols for more than two processes (using compositional generation)

lines); the generation of all protocols and Imcs for two (respectively, three) processes takes 5 (respectively, 40) minutes. This and all further experiments have been performed using a computer with a 2.8 GHz Intel® Xeon® processor and 148 GB RAM.

3. Analysis of Mutual Exclusion Protocols using CADP

This section is devoted to the automated analysis of the mutual exclusion protocols using the Cadp toolbox [18]. The protocols were analyzed by model checking and performance evaluation, both kinds of analysis being automated using Svl [43] scripts.[3] Therefore, we sought to be as generic as possible, in the sense that a property should not be tailored to a particular protocol, but rather be applicable to all protocols. This genericity comes at the price of additional transitions, such as "CS (Enter, i)" and "CS (Leave, i)" to delimit the critical sections precisely.

3.1. Model Checking

We expressed the correctness properties of the mutual exclusion protocols as formulas in the Mcl language [26], which extends the alternation-free μ-calculus [27] with regular expressions over transition sequences similar to those of Pdl [44], data-handling constructs inspired from functional programming languages, and (a generalization of) the infinite looping operator of Pdl-Δ [45]. Mcl enables a concise formulation of temporal properties, especially when these properties are parameterized by data values, such as the index of processes in mutual exclusion protocols. The Evaluator 4.0 model checker [26], built using the Open/Cæsar [46] graph exploration environment of Cadp, implements

[3]The Lotos NT specifications of the protocols and Svl scripts for state space generation, model checking, and performance evaluation will be included as a demo example in the next stable release of Cadp.

an efficient on-the-fly model checking procedure for MCL, by translating MCL formulas into boolean equation systems (BES) and solving them on the fly using the algorithms of the CÆSAR_SOLVE library [47]. The model checker also exhibits full diagnostics (examples and counterexamples) as subgraphs of the LTS illustrating the truth value of MCL formulas.

MCL is built from three kinds of formula. First, an *action formula* A characterizes actions (transition labels) of the LTS, which contain a gate name G followed by a list of values $v_1, ..., v_n$ exchanged during the rendezvous on G. An action formula is built from action patterns and the usual boolean connectors. An action pattern of the form "{G ?x:T !e **where** $b(x)$}" matches every action of the form "G v_1 v_2" where v_1 is a value of type T that is assigned to variable x, v_2 is the value obtained by evaluating the expression e, and the boolean expression $b(v_1)$ evaluates to true. Arbitrary combinations of value matchings ("!e") and value extractions ("?x:T") are allowed for matching actions containing several values. All variables assigned by value extraction are exported to the enclosing formula. Gate names G can also be extracted and manipulated as ordinary values of type String. The "?**any**" construct denotes a *wildcard* matching an arbitrary value regardless of its type. The clause "...", which may occur only once in an action pattern, specifies a list of zero or more wildcards.

Second, a *regular formula* R characterizes sequences of transitions in the LTS. A regular formula is built from action formulas and (extended) regular expression operators: concatenation ("$R_1.R_2$"), choice ("$R_1|R_2$"), unbounded iterations ("R^*" and "R^+"), iterations bounded by counters ("$R\{n\}$"), and a for-loop construct ("**for**").

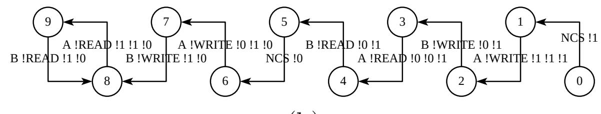
Third, a *state formula* F characterizes states of the LTS by specifying (finite or infinite) tree-like patterns going out from these states. A state formula is built from boolean connectors, possibility ("<R>F") and necessity ("[R]F") modalities containing regular formulas, minimal ("**mu** $X.F$") and maximal ("**nu** $X.F$") fixed point operators, quantifiers over finite domains ("**exists** x:$T.F$" and "**forall** x:$T.F$"), and the infinite looping operator ("<R>@"). An informal explanation of the semantics of MCL state formulas will be given by means of the examples below.

Mutual exclusion. This essential safety property of mutual exclusion protocols states that two processes can never execute simultaneously their critical section code. It can be expressed in MCL by a single box modality containing a regular formula that characterizes the undesirable sequences:

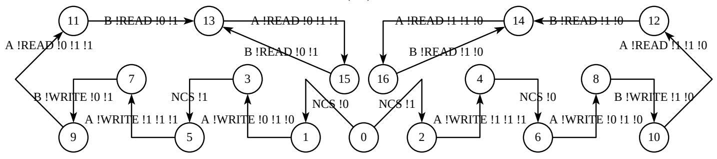
```
[ true* . { CS !"ENTER" ?i:Nat } . (not { CS !"LEAVE" !i })* .
  { CS !"ENTER" ?j:Nat where j <> i }
] false
```

This modality forbids the existence of sequences containing the entry of a process P_i in the critical section followed by the entry of another process P_j with $j \neq i$ in the critical section before P_i has left the critical section. Note how the process index i is extracted from a transition label by the first action pattern "{ CS !"ENTER" ?i:Nat }" and is used subsequently in the formula. The formula above does not make any assumption about the alternation of critical section entries and exits, but allows a process P_i to perform several "CS (Enter, i)" (resp. "CS (Leave, i)") actions while it is inside (resp. outside) its critical section. Under the additional hypothesis that every process executes its critical

```
loop
  non-critical section;
  A[i] := 1;
  B := j;
  while A[j] = 1
          and B = j do
  end while;
  critical section;
  A[i] := 0
end loop                (a)
```



(b)



(c)

Figure 3: (a) Peterson's protocol for process P_i in a configuration with two processes ($j = 1 - i$); (b) Livelock produced by spinning of process P_0 when process P_1 "has decided to stop"; (c) Livelocks produced after P_0 or P_1 crashed while executing their entry sections.

section entries and exits in a strict alternation, the formula above could be simplified as follows:

[**true*** . { CS !"ENTER" ... } . (**not** { CS !"LEAVE" ... })* .
  { CS !"ENTER" ... }
] **false**

which forbids the execution of two consecutive entries in critical section without an exit in between.

Livelock freedom. This liveness property[4] states that each time a process is in its entry section, then *some* process will eventually execute its critical section. A direct formulation of this property in MCL yields the formula below:

[**true*** . { NCS ?i:Nat } .
  (**not** { ?G:String ... !i **where** (G <> "NCS") **and** (G <> "CS") })* .
  { ?G:String ... !i **where** (G <> "NCS") **and** (G <> "CS") }
] **mu** X . (< **true** > **true and** [**not** { CS !"ENTER" ?**any** }] X)

The gate name extraction "?G:String" and the "**where** (G <> "NCS") **and** (G <> "CS")" clause specify the actions performed on shared variables, i.e., the actions other than accesses to critical and non critical sections. The minimal fixed point formula binding the X variable expresses the inevitable execution of some critical section after process P_i has executed the first read or write operation of its entry section. However, this formula is violated by all the protocols considered, because each time some process decides to stop its execution (an unrealistic hypothesis if we assume a fair scheduling of processes by the underlying operating system) the other processes can spin forever on reading

[4]Although some authors [30] use the term *deadlock* for this property, we prefer the term *livelock* used in [3] in order to avoid the confusion with the terminology used in model checking, in which deadlocks denote the sink states (i.e., without successors) of the LTS. Indeed, in a shared-memory setting without explicit locking operations, the behavior of the system cannot contain sink states, since each process can at any time execute some instruction.

shared variables. Figure 3(b) illustrates the counterexample of this formula exhibited by EVALUATOR 4.0 for Peterson's protocol with two processes. This protocol uses three shared variables, two of which are encoded as array cells $A[0]$, $A[1]$ and the third one by a separate variable B. The lasso-shaped diagnostic in Figure 3(b) shows that after process P_1 has executed its entry section and is ready to enter the critical section (because variable B has value 0) but does not do so, process P_0 may spin forever in the while loop of its entry section.

In fact, as stated in [30], a livelock situation occurs when all processes are cyclically executing at least one operation but none of them is able to progress towards its critical section. Therefore, an accurate formulation of livelock freedom in MCL must forbid the existence of such cycles:

```
[ true* . { NCS ?i:Nat } .
  (not { ?G:String ... !i where (G <> "NCS") and (G <> "CS") })* .
  { ?G:String ... !i where (G <> "NCS") and (G <> "CS") }
] not < for j:Nat from 0 to n − 1 do
           (not { CS ... })* . { ?G:String ... !j where G <> "CS" }
         end for
       > @
```

The "< ... > @" operator, which is the MCL counterpart of the infinite looping operator of PDL-Δ, expresses the existence of an infinite sequence consisting of the concatenation of subsequences satisfying a regular formula (note that, since the LTS is finite, any infinite sequence ends with a possibly non-elementary cycle). The "**for**" regular formula describes the repetition, for each $0 \leq j \leq n-1$ (where n is the number of processes) of a subsequence consisting of zero or more actions different from a critical section entry or exit, followed by one action, other than a critical section entry or exit, carried out by process P_j. In other words, the "**for**" regular formula specifies a subsequence of operations on shared variables containing (at least) one operation performed by each process in the system. Note that imposing a precise order for the occurrences of operations does not restrict the generality of the formula. Indeed, if there is some infinite sequence on which each process P_j can execute infinitely often at least one operation, then this sequence would also contain any ordering of these operations.

Starvation freedom. The absence of livelocks guarantees the global progress of the system, but does not ensure the access of individual processes to their critical sections. Starvation freedom is a stronger property (it implies livelock freedom), which states that each time a process is in its entry section, then *that* process will eventually execute its critical section. It can be expressed in MCL as follows:

```
[ true* . { NCS ?i:Nat } .
  (not { ?G:String ... !i where (G <> "NCS") and (G <> "CS") })* .
  { ?G:String ... !i where (G <> "NCS") and (G <> "CS") }
] not < for j:Nat from 0 to n − 1 do
           (not { CS ... !i })*
           { ?G:String ... !j where (j = i) implies (G <> "CS") }
         end for
       > @
```

The "< ... > @" operator describes a cycle containing at least one action performed by each process, but no access of process P_i to its critical section. To ensure the absence of starvation, a protocol must satisfy the necessary condition (stated in Section 2.1) that no process should remain in its critical section forever.

Bounded overtaking. Even if a mutual exclusion protocol is starvation-free, it is interesting to know, when a process P_i begins its entry section, how many times another process P_j can access its critical section before P_i enters its own critical section. This information can be determined using Evaluator 4.0 by checking the following Mcl formula for increasing values of *max*:

```
< true* . { NCS !i } .
   (not { ?G:String ... !i where (G <> "NCS") and (G <> "CS") })* .
   { ?G:String ... !i where (G <> "NCS") and (G <> "CS") } .
   (  for k:Nat from 0 to n − 1 do
          (not { CS ... !i })* . { ?G:String ... !k where (k = i) implies (G <> "CS") }
      end for .
      (not { CS ?any !i })* . { CS !"ENTER" !j }
   ) { max }
> true
```

This formula expresses the existence of a sequence in which process P_i executes its non-critical section, then the first instruction of its entry section, followed by *max* repetitions of a subsequence in which P_i executes some instruction but only P_j enters its critical section (a symmetric formula must be checked to determine the overtaking of process P_j by P_i). For each starvation-free protocol and couple of processes P_i, P_j, there exists a value of $max_{i,j}$ such that the formula above holds for $max_{i,j}$ and fails for $max_{i,j} + 1$. To minimize the number of model checking invocations, one can start with $max_{i,j} = 1$ and (if the formula holds for this value) keep doubling it until finding the first value $max'_{i,j}$ for which the formula fails, then use a dichotomic search to reduce the size of the interval $[max'_{i,j}/2, max'_{i,j}]$ to 1. In general, a starvation-free protocol may have different values of $max_{i,j}$ for different couples of processes P_i, P_j (as e.g., Szymanski's protocol), this fact indicating a lack of symmetry in the protocol.

First-Come, First-Served. Another way to estimate the quality of a mutual exclusion protocol w.r.t. the overtaking of processes is the *First-Come, First-Served* (Fcfs) property [48]. To formulate this property, one has to split the entry section of the protocol in two parts: a *doorway* section, consisting of a finite sequence of instructions, and a *waiting* section, consisting of an instruction sequence with unbounded length (which typically contains loops). The Fcfs property states that whenever a process P_i has executed its doorway section completely before another process P_j has executed the first instruction of its doorway section, then P_i will enter its critical section before P_j. For Peterson's protocol with two processes shown in Figure 3(a), the doorway section consists of the first two instructions of the entry section ("A[i] := 1; B := j") and the waiting section consists of the remaining loop of the entry section ("while A[j] = 1 and B = j do end while"). For this protocol, the Fcfs property can be expressed in Mcl using the formula below:

```
loop
  non-critical section;
  A[i] := 1;
  while A[j] = 1 do
     if B != i then
        A[i] := 0;
        while B != i do
        end while;
        A[i] := 1
     end if
  end while;
  critical section;
  B := j;
  A[i] := 0
end loop                (a)
```

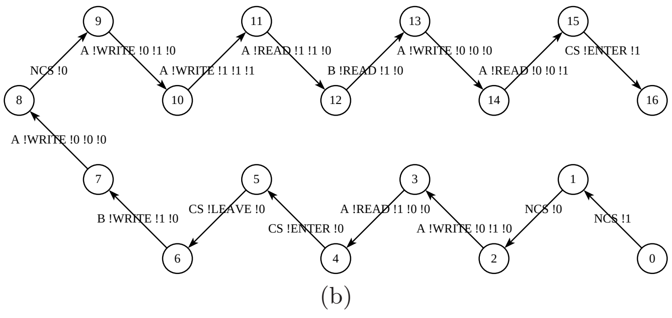


Figure 4: (a) Dekker's protocol for process P_i in a configuration with two processes ($j = 1 - i$); (b) Counterexample sequence showing that the protocol is not FCFS concerning process P_0.

```
[ true* . { NCS !j } . (not { ?G:String ... !j where G <> "CS" })* .
  { A !"WRITE" !i !1 !i } .
  (not ({ CS !"ENTER" !i } or { ?G:String ... !j where G <> "CS" }))* .
  { B !"WRITE" !j !i } .
  (not { CS !"ENTER" !i })* .
  { CS !"ENTER" !j }
] false
```

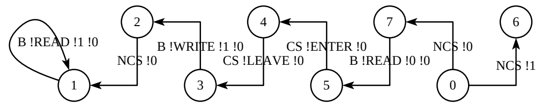
This safety property is satisfied by Peterson's protocol with two processes, which is therefore FCFS. As outlined in [48], any mutual exclusion protocol that is both livelock-free and FCFS is also starvation-free; however, not all starvation-free protocols are FCFS. For example, Dekker's protocol, shown in Figure 4(a), is not FCFS. The corresponding MCL formula, which considers Dekker's doorway section ("A[i] := 1"):

```
[ true* . { NCS !j } . (not { ?G:String ... !j where G <> "CS" })* .
  { A !"WRITE" !i !1 !i } .
  (not { CS !"ENTER" !i })* .
  { CS !"ENTER" !j }
] false
```

is violated by the protocol, as illustrated in Figure 4(b) by the shortest counterexample sequence exhibited by EVALUATOR 4.0. Note that the problematic execution of the doorway section by process P_0 is the second one in the sequence (the transition between states 9 and 10).

The FCFS property has a different shape for each protocol, since it depends on the partitioning of the entry section in doorway and waiting sections. We did not attempt in this study to characterize each protocol w.r.t. FCFS, but we considered instead the more generic measure of overtaking degree.

```
loop
  non-critical section;
  while B != i do
  end while;
  critical section;
  B := j
end loop              (a)
```



(b)

Figure 5: (a) Trivial one-bit protocol for process P_i ($j = 1 - i$); (b) Counterexample for the independent progress of P_0 when P_1 has stopped in its non-critical section.

Independent progress. A requirement formulated explicitly by Dijkstra [1] was that if a process stops (i.e., loops forever) in its *non*-critical section, this must not affect the access of the other processes to their critical sections. In subsequent works, this requirement is not mentioned as a property of mutual exclusion protocols, but is sometimes stated aside in the definition of the framework [3, 6]. However, we believe that this requirement is fundamental (at least from a model checking point of view) because it is not implied by the mutual exclusion and starvation freedom requirements, and therefore should be verified separately. In Mcl, it can be expressed using the following formula:

```
[ true* ] forall j:Nat among { 0 ... n − 1 } . (
   < { NCS !j } > true
   implies
   [ (not { ... !j })* ] forall i:Nat among { 0 ... n − 1 } . (
      (i <> j) implies < (not { ... !j })* > < { ... !i }* . { CS ... !i } > @
   )
)
```

which states that whenever a process P_j has stopped in its non-critical section (i.e., it is ready to execute the action "NCS (j)", but does not do so), then all the other processes P_i can access their critical section (modulo, of course, the mutual exclusion property stated previously).

To see that the property of independent progress is not implied by the two other properties of mutual exclusion protocols, consider the trivial one-bit protocol shown in Figure 5(a). This trivial protocol satisfies mutual exclusion and starvation freedom, but does not satisfy independent progress because it forces a strict alternation between the accesses of the two processes to their critical sections. The evaluation of the formula above on the Lts of the trivial protocol using Evaluator 4.0 yields the counterexample shown in Figure 5(b), in which process P_0 executes its main loop once but then spins forever in its entry section because P_1 has stopped in its non-critical section. The trivial protocol should be considered an unacceptable solution to the mutual exclusion problem, since it was proven in [6] (where independent progress is part of the framework definition) that any livelock-free mutual exclusion protocol must use at least *two* shared bits.

Finally, we can remark that the independent progress property cannot be made stronger without destroying the livelock or starvation freedom of the protocols: if a process is allowed to stop (e.g., by crashing) outside its non-critical section, then the other processes may spin forever without entering their critical sections. For all protocols considered here, we checked that this indeed holds; Figure 3(c) shows the diagnostic produced by EVALUATOR 4.0 illustrating, in the case of Peterson's protocol for two processes, the livelock of each process when the other one has crashed after executing the first instruction of its entry section.

Expressiveness and complexity. Regarding expressiveness, MCL lies between the $L\mu_1$ and $L\mu_2$ fragments of the modal μ-calculus, and is strictly more expressive than LTL [49], whose model checking problem can be translated into the evaluation of a single "`<` ... `>@`" operator encoding the underlying Büchi automaton [50]. Infinite looping operators "`<`R`>@`" whose regular subformulas R contain iteration operators (e.g., "$*$", "$+$", "**for**") belong to the fragment $L\mu_2$ of alternation depth two [27], because the maximal fixed points corresponding to the looping operators are mutually recursive with the minimal fixed points corresponding to the iteration operators of R. However, infinite looping operators can be evaluated in linear-time w.r.t. the size of the underlying BES (which is in turn proportional to the product of the LTS size and the formula size) by using the algorithm proposed in [26], which generalizes the detection of accepting cycles in Büchi automata. Note that this fact does not yield a linear-time model checking procedure for LTL, because the translations from LTL to μ-calculus [51] or to PDL-Δ [52] are not succinct.

Among the temporal properties shown above, several ones (livelock and starvation freedom, independent progress) are expressed as MCL formulas involving infinite looping operators. Livelock and starvation freedom (in their state-based variants) can be also expressed in LTL, by forbidding the existence of unfair cycles, as proposed in [30]. Independent progress can be also expressed in ACTL* [28] but not in LTL, because it states the existence of infinite sequences starting from several states of the LTS that are unknown in advance.

Model checking results. Tables 5 to 8 summarize the model checking results. In Tables 5 and 6, the first column gives the name of the protocol; the second and third columns indicate for which processes the protocol is livelock- and/or starvation-free; the fourth column indicates for which processes the protocol satisfies independent progress; the last columns give the maximal number of times process P_i can overtake process P_j in accessing the critical section (P_i/P_j). Table 7 contains the first to fourth columns for four processes; the remaining columns are given in Table 8.

Because the LTSs of the protocols for two processes are small, the execution of the SVL script (50 lines) implementing the model checking of all properties on all protocols takes about 5 minutes. For three (respectively, four) processes, the verification of all properties (including those for determining the degree of overtaking) on all protocols consists of more than 1000 (respectively, 2000) model checking tasks, which take altogether about 26 hours (respectively, 4 days, three of which are required for model checking the black-white bakery protocol), i.e., less than 2 (respectively, about 4) minutes per task on average.

All the protocols satisfy mutual exclusion, livelock freedom, as well as local deadlock freedom, i.e., the fact that in every state, each process can execute at least one action. All the protocols livelock if one of the processes crashes in its entry section, except the *test-and-set*, *test–test-and-set*, and the trivial protocol. Regarding the overtaking of processes, all starvation-free protocols (except Szymanski's and, for $n \geq 2$, Knuth's) are symmetric. For $n = 2$, the maximal degree (4) of overtaking is reached by Dekker's protocol. For $n =$

Protocol (2 processes)	Livelock-free	Starvation-free	Independent progress	Overtaking	
				P_0/P_1	P_1/P_0
Anderson	all	all	all	1	1
Burns & Lynch	all	P_0	all	∞	1
B&W Bakery	all	all	all	2	2
CLH	all	all	all	1	1
Dekker	all	all	all	4	4
Dijkstra	all	none	all	∞	∞
Kessels	all	all	all	2	2
Knuth	all	all	all	1	1
Lamport	all	none	all	∞	∞
MCS	all	all	all	1	1
Peterson	all	all	all	1	1
Peterson$_t$	all	all	all	1	1
Szymanski	all	all	all	2	1
tas	all	none	all	∞	∞
ttas	all	none	all	∞	∞
trivial	all	all	none	1	1
2b_p1	all	P_0	all	∞	1
2b_p2	all	P_0	all	∞	1
2b_p3	all	P_1	all	1	∞
3b_p1	all	all	all	2	2
3b_p2	all	P_0	all	∞	1
3b_c_p1_orig	all	all	all	1	1
3b_c_p1	all	all	all	1	1
3b_c_p2	all	all	all	1	1
3b_c_p3	all	all	all	1	1
4b_p1	all	P_0	all	∞	1
4b_p2	all	all	all	2	2
4b_c_p1	all	P_0	all	∞	1
4b_c_p2	all	P_1	all	1	∞

Table 5: Model checking results for two processes

3, the maximal degree (12) of overtaking is reached by the generalization of Peterson's protocol using a binary tree. For $n = 4$, the maximal degree (14) of overtaking is reached by Peterson's protocol. The unbounded overtaking of one process by another one has been checked by replacing, in the bounded overtaking formula given above, the bounded iteration operator "$R\{max\}$" by an infinite looping operator "`<` R `> @`". All livelock-free but not starvation-free protocols (except Dijkstra's, Lamport's, and the *test-and-set* protocol) are asymmetric w.r.t. overtaking, only one process being able to overtake all the others unboundedly.

3.2. Performance Evaluation

To measure the performance of a mutual exclusion protocol, we compute the throughput of the critical section, i.e., the steady state probability of being in the critical section. The higher the throughput, the more efficient the protocol, because the longer a process

Protocol (3 processes)	Livelock-free	Starv.-free	Indep. progress	Overtaking					
				P_0/P_1	P_0/P_2	P_1/P_0	P_1/P_2	P_2/P_0	P_2/P_1
Anderson	all	all	all	1	1	1	1	1	1
Burns & Lynch	all	P_0	all	∞	∞	1	∞	1	∞
B&W Bakery	all	all	all	2	2	2	2	2	2
Clh	all	all	all	1	1	1	1	1	1
Dijkstra	all	none	all	∞	∞	∞	∞	∞	∞
Knuth	all	all	all	1	2	2	1	1	2
Lamport	all	none	all	∞	∞	∞	∞	∞	∞
Mcs	all	all	all	1	1	1	1	1	1
Peterson	all	all	all	6	6	6	6	6	6
Peterson$_t$	all	all	all	1	1	1	1	12	12
Szymanski	all	all	all	2	2	1	2	1	1
tas	all	none	all	∞	∞	∞	∞	∞	∞
ttas	all	none	all	∞	∞	∞	∞	∞	∞
trivial	all	all	none	1	1	1	1	1	1

Table 6: Model checking results for three processes

Protocol (4 processes)	Livelock-free	Starvation-free	Independent progress
Anderson	all	all	all
Burns & Lynch	all	P_0	all
B&W Bakery	all	all	all
Clh	all	all	all
Dijkstra	all	none	all
Knuth	all	all	all
Lamport	all	none	all
Mcs	all	all	all
Peterson	all	all	all
Peterson$_t$	all	all	all
Szymanski	all	all	all
tas	all	none	all
ttas	all	none	all
trivial	all	all	none

Table 7: Model checking results for four processes (1/2)

Protocol	Overtaking											
(4 processes)	P_0/P_1	P_0/P_2	P_0/P_3	P_1/P_0	P_1/P_2	P_1/P_3	P_2/P_0	P_2/P_1	P_2/P_3	P_3/P_0	P_3/P_1	P_3/P_2
Anderson	1	1	1	1	1	1	1	1	1	1	1	1
Burns & Lynch	∞	∞	∞	1	∞	∞	1	∞	∞	1	∞	∞
B&W Bakery	2	2	2	2	2	2	2	2	2	2	2	2
CLH	1	1	1	1	1	1	1	1	1	1	1	1
Dijkstra	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
Knuth	1	2	4	4	1	2	2	4	1	1	2	4
Lamport	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
MCS	1	1	1	1	1	1	1	1	1	1	1	1
Peterson	14	14	14	14	14	14	14	14	14	14	14	14
Peterson$_t$	1	12	12	1	12	12	12	12	1	12	12	1
Szymanski	2	2	2	1	2	2	1	1	2	1	1	1
tas	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
ttas	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
trivial	1	1	1	1	1	1	1	1	1	1	1	1

Table 8: Model checking results for four processes (2/2)

is in the critical section, the less time it spends executing the protocol or waiting to enter the critical section.

Methodology and parameter setting. Performance evaluation of an IMC is based on the transformation of the IMC into a *Continuous-Time Markov Chain* (CTMC) extended with probabilistic choices. This consists of three steps:

- A first step is to transform the IMC into a stochastic LTS by renaming all actions: (1) each action not representing a delay is hidden, i.e., renamed into the invisible action (written `i` in LOTOS NT and CADP), then (2) each `MU` action is transformed into an exponential delay by associating a rate λ to it, i.e., renaming it into "`rate` λ". Our use of exponential delays reflects the fact that we make hypotheses only about the *relations* between the mean values of the actual durations, because our model-based performance evaluation does assume neither a particular application nor a particular hardware architecture.

  In all our experiments, we keep the rates for accesses to the shared variables constant: each read access has a rate of 3000 and each write access has a rate of 2000, reflecting that, on average, a write access is generally slower than a read access. For complex operations, namely *fetch-and-store* (used by the protocols CLH and MCS), *compare-and-swap* (used by the protocols MCS, *test-and-set* and *test–test-and-set*), and *read-and-increment* (used by Anderson's protocol), we used a rate corresponding to a read access followed by a write access, reflecting that these operations have first to read the memory and then to write it. We assume that reading from a local cache is 50 times faster than reading from global memory, and that writing to a local cache is 10% slower than reading. We also assume a write-back policy, i.e., a write into a local cache is not immediately propagated to global memory. Thus, we used the following rates:

    - read from a local cache: $150,000$,
    - read from global memory: 3000,
    - read from a remote cache: 1200 (because the variable must first be written to global memory, and than read from global memory),
    - write to a local cache: $135,000$,
    - write to global memory: 2000 (also in the case that another cache must be invalidated),
    - complex operation (i.e., *fetch-and-store*, *compare-and-swap*, and *read-and-increment*) in a local cache: $71,052$ (because the value has to be read and written),
    - complex operation on a variable present in several caches: 1200 (because the value has to be read and written), and
    - complex operation on a variable present in another cache: 750 (because the variable must first be written to global memory, and then read and written).

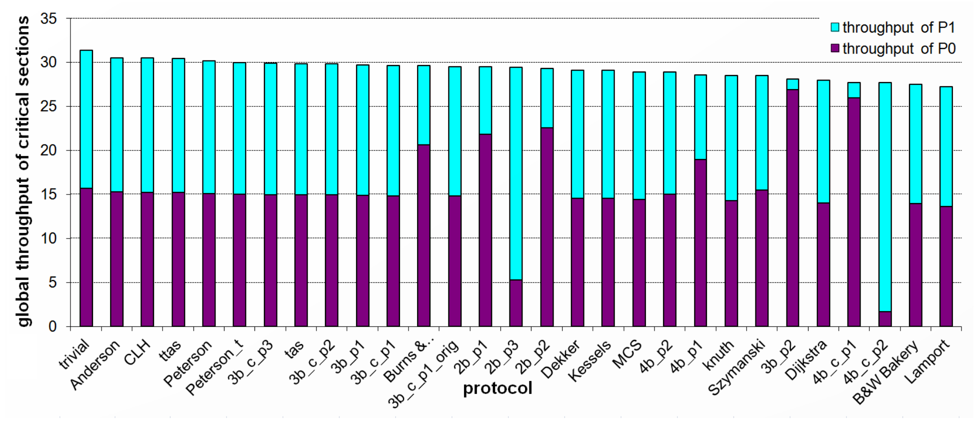
  If not specified otherwise, we use the rate 100 for the critical section, i.e., making the assumption that the critical section contains (on average) several accesses to shared data. Hence, to compare the protocols in different usage scenarios, we vary mainly the delay for the non-critical section.

- In a second step, the nondeterminism is solved as discussed in Section 2.3 by assuming a uniform scheduler.[5] Practically, each nondeterministic choice is replaced by a uniform probabilistic choice, by renaming all i transitions into "prob $1/n$".
- Finally, we compute the throughput of the entries into the critical sections by all processes in the steady state using the BCG_STEADY tool [22], which is able to handle CTMCs extended with probabilistic choices, and, for large CTMCs, the CUNCTATOR on-the-fly steady state simulator [23], which provides a uniform scheduler to solve nondeterministic choices.

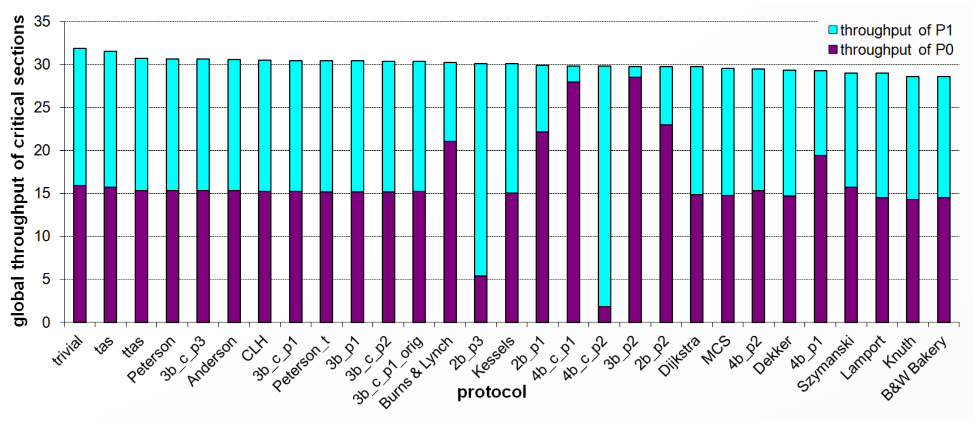
The results of our experiments are shown in Figures 6 to 10. Because these figures depend on the chosen rates, the concrete values and rankings are less interesting than the relations and tendencies when varying different parameters. Also, the long critical and non-critical sections (that are the same for all protocols) mask to some extent differences between protocols, especially for few processes, because each process spends most of its time in the critical and non-critical section.

The performance evaluation experiments are automated by an SVL script (400 lines), which, for each experiment, computes the appropriate rates, renames the transition labels, and then computes the throughputs in the steady state.

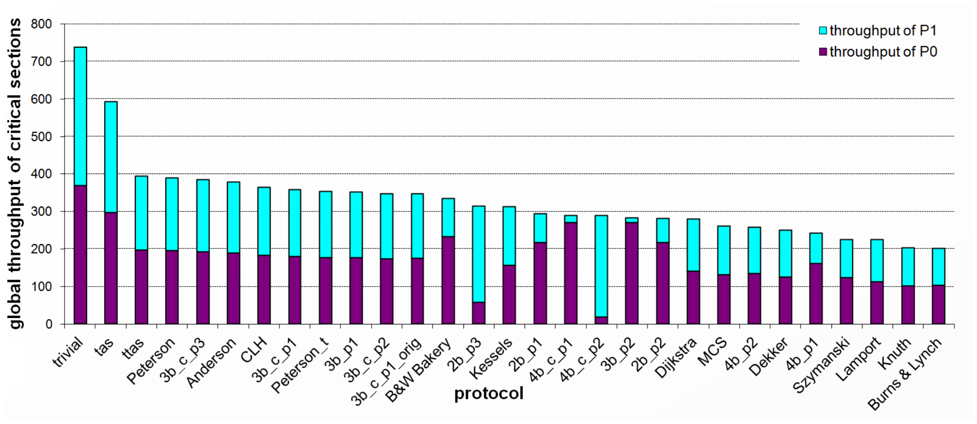
[5]In [29], each renamed IMC is minimized for stochastic branching bisimulation [53] before solving the remaining nondeterminism. We no longer perform this additional step, because it is incorrect for more than two processes: indeed, it does not accumulate the probabilities when eliminating some, but not all, internal transitions going out of a state (a situation that did not occur for the case of two processes considered in [29]).



(a) Global throughput without caching



(b) Global throughput with caching



(c) Global throughput with caching and very short critical section

Figure 6: Throughput for two processes (critical section two times faster than non-critical section)

Steady-state analysis for two processes. For two processes, we compute all performance measures in five minutes using Bcg_Steady.

Figure 6 shows the throughputs of all protocols for two processes, using 50 for the rate of the non-critical section (thus, the non-critical section is, on average, two times longer than the critical section). Figure 6(a) shows the throughputs without considering caches. As shown in Figure 6(b), taking caching into account (slightly) improves the performance of the protocols, but also changes the ranking of the protocols concerning their performance.[6] Figure 6(c) shows the throughputs with caching and a very short critical section (rate $10,000,000$), an unrealistic situation that stresses the differences in the performance of the protocols: interestingly, this has no influence on the ranking. Therefore, in the subsequent experiments we keep the more realistic rates, even though the differences between the protocols are somewhat covered by the long critical and non-critical sections.

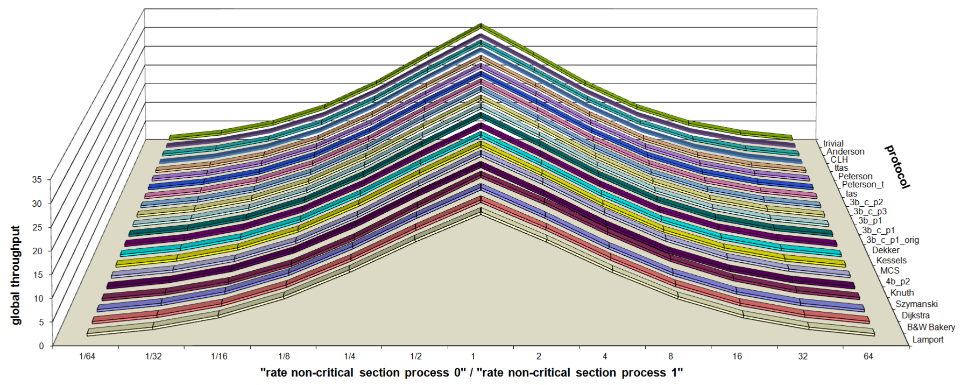
One observes significant differences in the throughputs of the two processes if and only if the protocol is asymmetric, i.e., when one process can overtake more often than it can be overtaken by the other process; for these protocols, the qualitative and quantitative properties are related in the sense that the process that can overtake the other has a significantly higher throughput. For symmetric protocols, the throughput of process P_0 is half the global throughput. A second observation is that the complexity of the protocol (number of shared variables and length of entry and exit sections) impacts its performance: for instance, the most complex protocol (black-white bakery) is among the least efficient, whereas the trivial one-bit protocol is the most efficient.

We also observe that making a protocol symmetric might (slightly) improve its performance. For instance, the original version of the automatically generated protocol 3b_c_p1 as described in [30] is asymmetric: with the same rates as in Figure 6(a), the throughput of process P_0 (14.86) is higher than the throughput of process P_1 (14.67). However, the symmetric version of this protocol has a higher global throughput (29.65 instead of 29.535, i.e., an increase of less than 0.4%, to be compared with the 15% performance difference between the least and most efficient protocol).

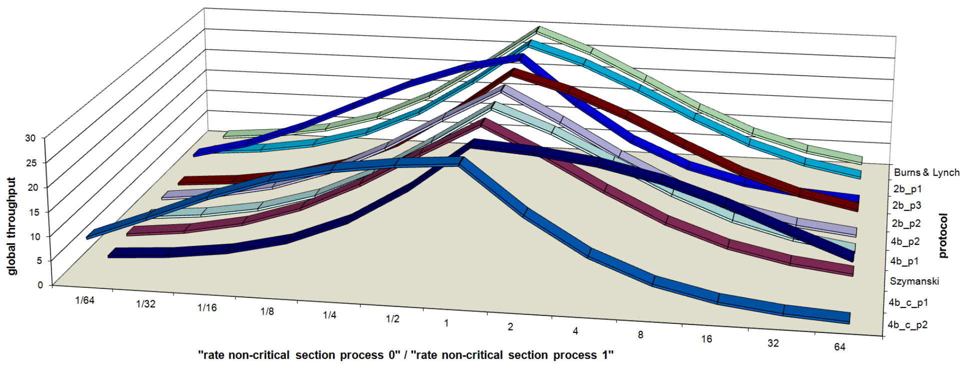
The three plots of Figure 7 show the effect of varying the ratio between the non-critical section rates of the two processes. In all three plots, for ratio 1, the rate of the non-critical section is 50 for both processes; towards the left, process P_0 is slowed down (by decreasing the rate of the non-critical section of P_0); towards the right, process P_1 is slowed down (by decreasing the rate of the non-critical section of P_1).

Figure 7(a) graphically justifies the term "symmetric" protocols: they are symmetric in the sense that slowing down process P_0 has exactly the same effect on the global throughput as slowing down process P_1: in both cases the general throughput decreases in the same way. Figure 7(b) shows that the situation is different for asymmetric protocols: slowing down the advantaged process that can overtake the other one reduces the general throughput more than slowing down the disadvantaged process that can be overtaken. This seems intuitive, because slowing down the advantaged process slows down both processes, whereas slowing down the disadvantaged process should not overly impact the advantaged process. Figure 7(c) confirms this intuition. On the one hand, for all those asymmetric protocols where process P_0 can overtake process P_1 infinitely, slowing down

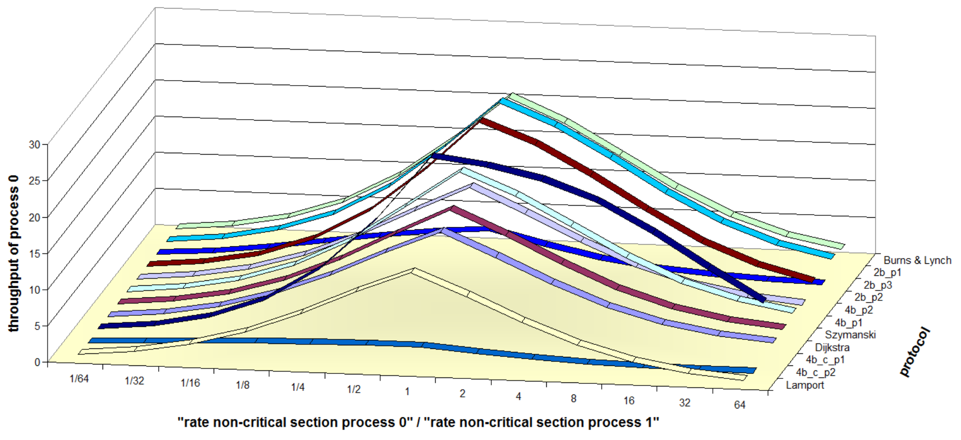
[6]Experiments with further increasing the speed-up of cache accesses with respect to an access to global memory did not lead to a ranking different from the one shown in Figure 6(b).



(a) Global throughput for symmetric protocols



(b) Global throughput for asymmetric protocols



(c) Throughput of process P_0 for asymmetric protocols

Figure 7: Performance when varying the ratio ncs-rate-p$_0$/ncs-rate-p$_1$

P_1 has less impact on the throughput of P_0 than slowing down P_0. On the other hand, for the two protocols 2b_p3 and 4b_c_p2, where P_0 can be overtaken infinitely by P_1, slowing down P_1 has more impact on the throughput of P_0 than slowing down P_0.

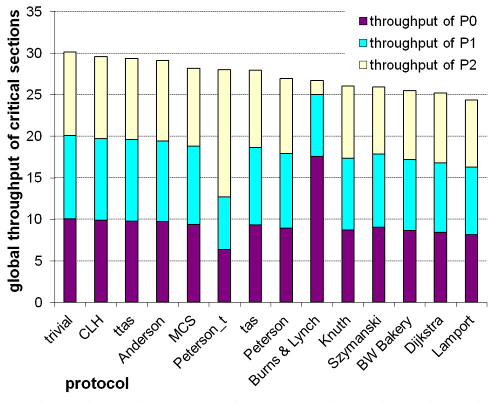
Steady-state analysis for more than two processes. For some protocols, we also compute the performance for more than two processes. For three processes, we compute all performance measures in 47 hours (43 of which are required for the black-white bakery protocol) using BCG_STEADY. For more processes, we use CUNCTATOR with three seeds for the random number generator: executing ten million steps of all protocols requires 5.5 hours for four processes, 21.5 hours for five processes, and about three weeks for ten processes. The standard deviation of the three numbers computed by CUNCTATOR is less than 0.15 (curiously, for five processes, even less than 0.1), i.e., the results can be considered accurate up to 0.3. The precision is slightly better in the case without caching; an explanation might be numerical stability due to closer transition rates than in the case with caching. For four (respectively, five) processes, simulating fifty million steps reduces the standard deviation to below 0.05 (respectively, 0.055). Concerning the trade-off between precision and execution time, it seems reasonable to switch from BCG_STEADY to CUNCTATOR when passing from three to four processes. On the one hand, for three processes, using CUNCTATOR requires more time to yield less precise results[7]. On the other hand, for four processes, using BCG_STEADY requires too much time (e.g., for the Dijkstra's and the black-white bakery protocol, BCG_STEADY did not manage to compute the result in five months).

The results of these experiments are shown in Figures 8 and 9, for all of which we use the same rates as for Figures 6(a) (without caching) and 6(b) (with caching). A first observation is that increasing the number of processes highlights performance differences between the protocols: in the case without caching, the difference between the throughputs of the fastest and slowest protocol almost doubles when passing from two processes (4.07) to four processes (7.72). A second observation is that caching improves the performance of the protocols, independently of the number of processes.

A third observation is that most protocols are equitable, in the sense that all processes have similar throughputs (for three processes, strict equality holds for the protocols CLH, Knuth, MCS, *test-and-set*, *test–test-and-set*, and trivial). A notable exception is the protocol proposed by Burns & Lynch [6], which clearly favors processes with lower indices. Interestingly, although functional verification (i.e., Table 6) shows that, theoretically, process P_2 can overtake process P_1 infinitely often, this seems rarely to occur in practice, because the throughput of P_2 (1.9 in the case with caching) is significantly lower than that of P_1 (7.98 in the case with caching).

Comparison to experimental performance measures. Comparing our results with experimental performance measures found in the literature is not straighforward, because, on the one hand, the precise delays of all different operations are not available, and on the other hand, we do not have access to the particular architectures used in the experiments. Furthermore, these experiments usually measure the overall execution time of a set of processes on a given multiprocessor architecture, whereas we compute the throughput

[7]For three processes, to execute one billion steps for all protocols takes more than a week; the standard deviation of the computed numbers is less than 0.001.


(a) Global throughput without caching

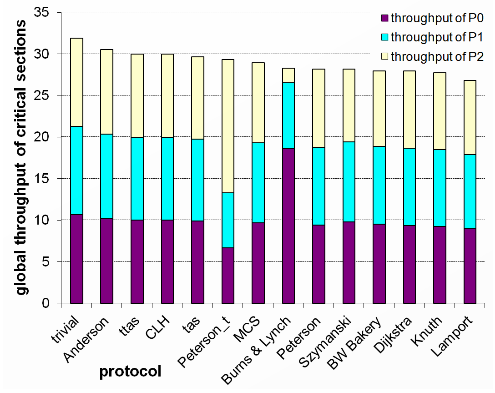
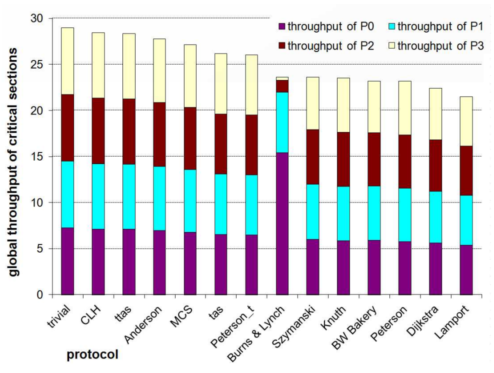

(b) Global throughput with caching

Figure 8: Throughput for three processes (critical section two times faster than non-critical section)


(a) Global throughput without caching

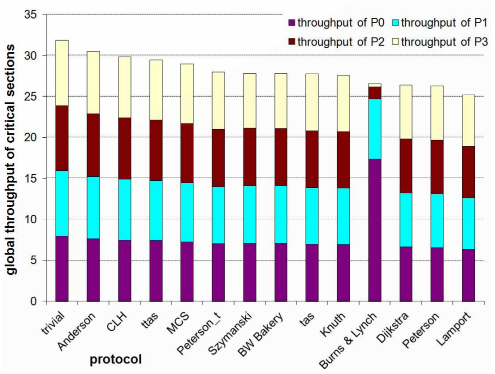
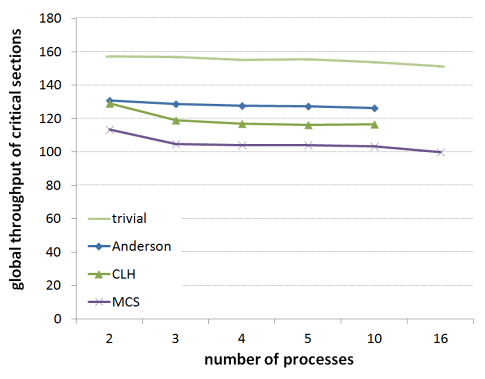

(b) Global throughput with caching

Figure 9: Throughput for four processes (critical section two times faster than non-critical section)


(a) Scalable protocols

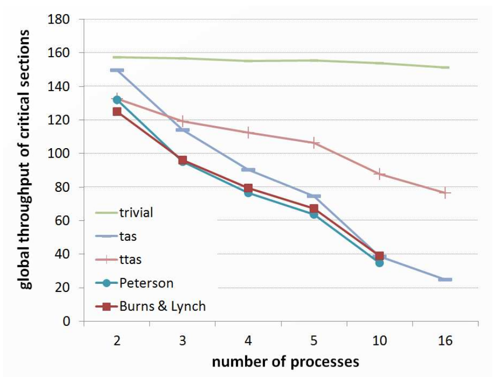

(b) Unscalable protocols

Figure 10: Global throughput for an increasing number of processes

of the critical section. Nevertheless, to relate the tendencies computed for our models with the measurements of executions on particular architectures, we computed the global throughput for some protocols for up to 16 processes, the results of which are shown in Figure 10. For these computations, we used rates inspired by the experiments of [32], i.e., a rather short critical section (corresponding to the increment of a shared counter) and a non-critical section five times as long as the critical section.

Figure 10(a) shows that the more recent protocols making use of complex operations scale well in the sense that there is no significant drop in the global throughput when increasing the number of processors. However, Figure 10(b) shows that other protocols (including *test-and-set* and *test–test-and-set*) do not scale so well: increasing the number of processes yields a significant drop in the global throughput of the critical section, confirming the observation of [32], which led to the development of the array-based queue lock.

Notice that these tendencies are also present, even if not so prominent, in Figures 6, 8(b), and 9(b), which show that, when increasing the number of processors from two to four, the global throughput of *test–test-and-set* drops from 30.7 to 29.5, whereas the global throughput of *test-and-set* drops from 31.5 to 27.7.

4. Conclusion and Future Work

This study illustrates a model-based approach, supported by the Cadp toolbox, for analyzing the functional behavior and performance of shared-memory mutual exclusion protocols. As the underlying semantic model, we used Imcs [17], which provide a uniform framework suitable both for model checking and performance evaluation. We carried out the analysis of 27 protocols using the state-of-the-art functionalities provided by the Cadp toolbox [18]: formal specification using the Lotos NT imperative-style process-algebraic language; description of functional properties using the Mcl data-based temporal language; steady-state analysis of Imcs using the Bcg_Steady and Cunctator tools; and automation of the analysis procedures using Svl scripts.

We attempted to formulate the correctness properties of mutual exclusion protocols accurately and observed that several of them (livelock and starvation freedom, independent progress, unbounded overtaking) belong to $L\mu_2$, the μ-calculus fragment of alternation depth two; however, they can still be expressed using the infinite looping operator of Pdl-Δ [45], which can be checked in linear-time [26]. Performance evaluation made it possible to compare the protocols according to their efficiency (global and individual throughput of processes) and to study the effect of varying several parameters (relative speeds of processes, ratio between the time spent in critical and non-critical sections, etc.). We observed that symmetric protocols are more robust concerning the difference in execution speed between processes, which confirms the importance of the symmetry requirement originally formulated by Dijkstra [1]. The quantitative results were corroborated by those of functional verification, in particular the presence of (asymmetric) starvation of processes, detected using temporal formulas, was clearly reflected in the steady-state behavior of the corresponding protocols. We also studied the performance of mutual exclusion protocols involving $n > 2$ processes, which so far were subject only to analytical studies [54].

Concerning future work, we plan to improve Bcg_Steady by optimizing the internal representation of sparse matrices; preliminary tests indicate a significant reduction of execution time. It would also be interesting to study the effect of the degree of contention (i.e., the number of processes trying to enter the critical section at the same time) on the performance of mutual exclusion protocols, as some of them (e.g., [8]) are specifically optimized for this situation. Finally, our models could be extended to take into account different assumptions, such as other cache-coherency policies (possibly allowing inconsistent views of the memory) or a number of processors strictly smaller than the number of processes.

References

[1] E. W. Dijkstra, Solution of a problem in concurrent programming control, Communications of the ACM 8 (9) (1965) 569–570.

[2] M. Raynal, Algorithmique du parallélisme : le problème de l'exclusion mutuelle, Dunod-Informatique, Paris, 1984.

[3] J. H. Anderson, Y.-J. Kim, T. Herman, Shared-memory mutual exclusion: Major research trends since 1986, Distributed Computing 16 (2003) 75–110.

[4] G. Taubenfeld, Synchronization Algorithms and Concurrent Programming, Pearson, Prentice Hall, 2006.

[5] D. E. Knuth, Additional comments on a problem in concurrent programming control, Communications of the ACM 9 (5) (1966) 321–322.

[6] J. E. Burns, N. A. Lynch, Mutual exclusion using indivisible reads and writes, in: Proceedings of the 18th Annual Allerton Conference on Communication, Control, and Computing, 1980, pp. 833–842.

[7] G. L. Peterson, Myths about the mutual exclusion problem, Information Processing Letters 12 (3) (1981) 115–116.

[8] L. Lamport, A fast mutual exclusion algorithm, ACM Transactions on Computer Systems 5 (1) (1987) 1–11.

[9] B. K. Szymanski, A simple solution to Lamport's concurrent programming problem with linear wait, in: Proceedings of the 2nd International Conference on Supercomputing ICS'88 (Saint Malo, France), ACM Computer Society Press, 1988, pp. 621–626.

[10] G. Taubenfeld, The black-white bakery algorithm and related bounded-space, adaptive, local-spinning and FIFO algorithms, in: R. Guerraoui (Ed.), Proceedings of the 18th International Conference on Distributed Computing DISC'04 (Amsterdam, The Netherlands), Vol. 3274 of Lecture Notes in Computer Science, Springer-Verlag, 2004, pp. 56–70.

[11] Z. Manna, A. Pnueli, Tools and Rules for the Practicing Verifier, ACM Press and Addison-Wesley, 1991, pp. 125–159.

[12] H. E. Jensen, N. A. Lynch, A proof of Burns n-process mutual exclusion algorithm using abstraction, in: B. Steffen (Ed.), Proceedings of the 4th International Conference on Tools and Algorithms for the Construction and Analysis of Systems TACAS'98 (Lisbon, Portugal), Vol. 1384 of Lecture Notes in Computer Science, Springer-Verlag, Berlin, 1998, pp. 409–423.

[13] G. Delzanno, A. Podelski, Model checking in CLP, in: R. Cleaveland (Ed.), Proceedings of the 5th International Conference on Tools and Algorithms for the Construction and Analysis of Systems TACAS'99 (Amsterdam, The Netherlands), Vol. 1579 of Lecture Notes in Computer Science, Springer-Verlag, Berlin, 1999, pp. 223–239.

[14] M. Botincan, ASML specification and verification of Lamport's bakery algorithm, Journal of Computing and Information Technology 13 (4) (2005) 313–319.

[15] J.-H. Yang, J. H. Anderson, A fast, scalable mutual exclusion algorithm, Distributed Computing 9 (1) (1995) 51–60.

[16] X. Zhang, Y. Yan, R. Castaneda, Evaluating and designing software mutual exclusion algorithms on shared-memory multiprocessors, IEEE Parallel Distributed Technology 4 (1) (1996) 25–42.

[17] H. Hermanns, Interactive Markov Chains and the Quest for Quantified Quality, Vol. 2428 of Lecture Notes in Computer Science, Springer-Verlag, 2002.

[18] H. Garavel, F. Lang, R. Mateescu, W. Serwe, CADP 2010: A toolbox for the construction and analysis of distributed processes, in: P. A. Abdulla, K. R. M. Leino (Eds.), Proceedings of the 17th International Conference on Tools and Algorithms for the Construction and Analysis of Systems TACAS'2011 (Saarbrücken, Germany), Vol. 6605 of Lecture Notes in Computer Science, Springer-Verlag, 2011, pp. 372–387.

[19] H. Hermanns, J.-P. Katoen, Performance evaluation : = (process algebra + model checking) Markov chains, in: K. G. Larsen, M. Nielsen (Eds.), Proceedings of the 12th International Conference on Concurrency Theory CONCUR'01 (Aalborg, Denmark), Vol. 2154 of Lecture Notes in Computer Science, Springer Verlag, 2001, pp. 59–81.

[20] D. Champelovier, X. Clerc, H. Garavel, Y. Guerte, C. McKinty, V. Powazny, F. Lang, W. Serwe, G. Smeding, Reference manual of the LOTOS NT to LOTOS translator (version 5.2), INRIA/VASY, 117 pages (Feb. 2011).

[21] H. Garavel, M. Sighireanu, Towards a second generation of formal description techniques – rationale for the design of E-LOTOS, in: J.-F. Groote, B. Luttik, J. Wamel (Eds.), Proceedings of the 3rd International Workshop on Formal Methods for Industrial Critical Systems FMICS'98 (Amsterdam, The Netherlands), CWI, Amsterdam, 1998, pp. 187–230, invited lecture.

[22] H. Hermanns, C. Joubert, A set of performance and dependability analysis components for CADP, in: H. Garavel, J. Hatcliff (Eds.), Proceedings of the 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems TACAS'2003 (Warsaw, Poland), Vol. 2619 of Lecture Notes in Computer Science, Springer-Verlag, 2003, pp. 425–430.

[23] N. Coste, H. Garavel, H. Hermanns, F. Lang, R. Mateescu, W. Serwe, Ten years of performance evaluation for concurrent systems using CADP, in: T. Margaria, B. Steffen (Eds.), Proceedings of the 4th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation, ISoLA'2010 (Heraclion, Crete), Part II, Vol. 6416 of Lecture Notes in Computer Science, Springer-Verlag, 2010, pp. 128–142.

[24] G. Chehaibar, M. Zidouni, R. Mateescu, Modeling multiprocessor cache protocol impact on MPI performance, in: Proceedings of the 2009 IEEE International Workshop on Quantitative Evaluation of Large-Scale Systems and Technologies QuEST'09 (Bradford, UK), IEEE Computer Society Press, 2009.

[25] H. Garavel, H. Hermanns, On combining functional verification and performance evaluation using CADP, in: L.-H. Eriksson, P. A. Lindsay (Eds.), Proceedings of the 11th International Symposium of Formal Methods Europe FME'2002 (Copenhagen, Denmark), Vol. 2391 of Lecture Notes in Computer Science, Springer-Verlag, 2002, pp. 410–429, full version available as INRIA Research Report 4492.

[26] R. Mateescu, D. Thivolle, A model checking language for concurrent value-passing systems, in: J. Cuellar, T. Maibaum, K. Sere (Eds.), Proceedings of the 15th International Symposium on Formal Methods FM'08 (Turku, Finland), no. 5014 in Lecture Notes in Computer Science, Springer-Verlag, 2008, pp. 148–164.

[27] E. A. Emerson, C.-L. Lei, Efficient model checking in fragments of the propositional mu-calculus, in: Proceedings of the 1st LICS, 1986, pp. 267–278.

[28] R. D. Nicola, F. W. Vaandrager, Action versus State Based Logics for Transition Systems, Vol. 469 of Lecture Notes in Computer Science, Springer Verlag, 1990, pp. 407–419.

[29] R. Mateescu, W. Serwe, A study of shared-memory mutual exclusion protocols using CADP, in: S. Kowalewski, M. Roveri (Eds.), Proceedings of the 15th International Workshop on Formal Methods for Industrial Critical Systems FMICS'2010 (Antwerp, Belgium), Vol. 6371 of Lecture Notes in Computer Science, Springer-Verlag, 2010, pp. 180–197.

[30] Y. Bar-David, G. Taubenfeld, Automatic discovery of mutual exclusion algorithms, in: F. E. Fich (Ed.), Proceedings of the 17th International Conference on Distributed Computing DISC'03 (Sorrento, Italy), Vol. 2848 of Lecture Notes in Computer Science, Springer-Verlag, 2003, pp. 136–150.

[31] ISO/IEC, Enhancements to LOTOS (E-LOTOS), International Standard 15437:2001, International Organization for Standardization — Information Technology, Genève (Sep. 2001).

[32] T. E. Anderson, The performance of spin lock alternatives for shared memory multiprocessors, IEEE Transactions on Parallel and Distributed Systems 1 (1) (1990) 6–16.

[33] T. S. Craig, Building FIFO and priority-queuing spin locks from atomic swap, Tech. Rep. 93-02-02, Department of Computer Science and Engineering, University of Washington, Seattle (Feb. 1993).

[34] P. S. Magnusson, A. Landin, E. Hagersten, Queue locks on cache coherent multiprocessors, in: H. J. Siegel (Ed.), Proceedings of the 8th International Parallel Processing Symposium IPPS (Cancún, Mexico), IEEE Computer Society Press, 1994, pp. 165–171.

[35] E. W. Dijkstra, Co-operating Sequential Processes, Academic Press, New York, 1968, pp. 43–112.

[36] J. L. W. Kessels, Arbitration without common modifiable variables, Acta Informatica 17 (1982) 135–141.

[37] J. M. Mellor-Crummey, M. L. Scott, Algorithms for scalable synchronization on shared-memory multiprocessors, ACM Transactions on Computer Systems 9 (1) (1991) 21–65.

[38] M. S. Papamarcos, J. H. Patel, A low-overhead coherence solution for multiprocessors with private

cache memories, in: Proceedings of the 11th International Symposium on Computer Architecture, 1984.

[39] A. Shiryaev, Probability, 2nd Edition, Vol. 95 of Graduate Texts in Mathematics, Springer-Verlag, Berlin, 1996.

[40] M. L. Puterman, Markov Decision Processes: Discrete Stochastic Dynamic Programming, Wiley, 1994.

[41] L. Zhang, M. R. Neuhäußer, Model checking interactive Markov chains, in: J. Esparza, R. Majumdar (Eds.), Proceedings of the 16th International Conference on Tools and Algorithms for the Construction and Analysis of Systems TACAS 2010 (Paphos, Cyprus), Vol. 6015 of Lecture Notes in Computer Science, Springer-Verlag, Berlin, 2010, pp. 53–68.

[42] J.-P. Krimm, L. Mounier, Compositional state space generation from LOTOS programs, in: E. Brinksma (Ed.), Proceedings of TACAS'97 Tools and Algorithms for the Construction and Analysis of Systems (University of Twente, Enschede, The Netherlands), Vol. 1217 of Lecture Notes in Computer Science, Springer-Verlag, Berlin, 1997, extended version with proofs available as Research Report VERIMAG RR97-01.

[43] H. Garavel, F. Lang, SVL: a scripting language for compositional verification, in: M. Kim, B. Chin, S. Kang, D. Lee (Eds.), Proceedings of the 21st IFIP WG 6.1 International Conference on Formal Techniques for Networked and Distributed Systems FORTE'2001 (Cheju Island, Korea), IFIP, Kluwer Academic Publishers, 2001, pp. 377–392, full version available as INRIA Research Report RR-4223.

[44] M. J. Fischer, R. E. Ladner, Propositional dynamic logic of regular programs, Journal of Computer and System Sciences (18) (1979) 194–211.

[45] R. Streett, Propositional dynamic logic of looping and converse, Information and Control (54) (1982) 121–141.

[46] H. Garavel, Open/Cæsar: An open software architecture for verification, simulation, and testing, in: B. Steffen (Ed.), Proceedings of the First International Conference on Tools and Algorithms for the Construction and Analysis of Systems TACAS'98 (Lisbon, Portugal), Vol. 1384 of Lecture Notes in Computer Science, Springer-Verlag, Berlin, 1998, pp. 68–84, full version available as INRIA Research Report RR-3352.

[47] R. Mateescu, Caesar_solve: A generic library for on-the-fly resolution of alternation-free boolean equation systems, Springer International Journal on Software Tools for Technology Transfer (STTT) 8 (1) (2006) 37–56, full version available as INRIA Research Report RR-5948, July 2006.

[48] M. Herlihy, N. Shavit, The Art of Multiprocessor Programming, Elsevier, 2008.

[49] Z. Manna, A. Pnueli, The Temporal Logic of Reactive and Concurrent Systems, Vol. I (Specification), Springer-Verlag, 1992.

[50] E. Clarke, O. Grumberg, D. Peled, Model Checking, MIT Press, 2000.

[51] M. Dam, CTL* and ECTL* as fragments of the modal μ-calculus, Theoretical Computer Science 126 (1) (1994) 77–96.

[52] P. Wolper, A translation from full branching time temporal logic to one letter propositional dynamic logic with looping, unpublished manuscript (1982).

[53] H. Hermanns, M. Siegle, Bisimulation algorithms for stochastic process algebras and their BDD-based implementation, in: J.-P. Katoen (Ed.), Proceedings of the 5th International AMAST Workshop ARTS'99 (Bamberg, Germany), Vol. 1601 of Lecture Notes in Computer Science, Springer-Verlag, 1999, pp. 244–265.

[54] J. H. Anderson, Y.-J. Kim, Adaptive mutual exclusion with local spinning, in: Proceedings of the 14th International Symposium on Distributed Computing, Vol. 1914 of Lecture Notes in Computer Science, Springer-Verlag, Berlin, 2000, pp. 29–43.