HAL
open science

A Study of Shared-Memory Mutual Exclusion Protocols using CADP

Radu Mateescu, Wendelin Serwe

To cite this version:

Radu Mateescu, Wendelin Serwe. A Study of Shared-Memory Mutual Exclusion Protocols using CADP. 15th International Workshop on Formal Methods for Industrial Critical Systems FMICS'2010, Sep 2010, Antwerp, Belgium. inria-00532897

HAL Id: inria-00532897

https://inria.hal.science/inria-00532897

Submitted on 4 Nov 2010

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

A Study of Shared-Memory Mutual Exclusion Protocols using CADP

Radu Mateescu and Wendelin Serwe

Inria Grenoble – Rhône-Alpes, Inovallée, 655, av. de l'Europe,
Montbonnot, F-38334 Saint Ismier, France
{Radu.Mateescu,Wendelin.Serwe}@inria.fr

Abstract. Mutual exclusion protocols are an essential building block of concurrent 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 in the shared-memory setting, 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 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, whose accurate formulation as temporal logic formulas in the action-based setting turns out to be quite involved.

1 Introduction

Mutual exclusion is a long-standing problem in concurrent programming, formulated initially by Dijkstra almost half a century ago [10]. 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 system is guaranteed to progress. 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 [37, 2, 42]). Most of the effort has been concentrated on analyzing the functional correctness of these protocols, either by hand-written proofs [10, 26, 5, 35, 27, 40, 2, 41] or by applying automated reasoning and model checking techniques [29, 24, 9, 4]. However, much less attention has been given to the model-based performance evaluation of these protocols, most of the existing works dealing with performance measurements of protocol implementations on specific architectures [43, 45].

In this paper, we show how Interactive Markov Chains (Imc) [19] and their implementation in the Cadp verification toolbox [17] can be applied to the 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 [21], which can be employed as precise approximations of arbitrary (discrete or continuous) probability distributions.

As high-level specification language for IMCs, we use LOTOS NT [6, 18], 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 [20] tool of CADP, which computes the throughputs of various actions at steady-state. This allows to compare the performance of various protocols and to study the impact of certain parameters (e.g., 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 [7]. For cache-coherent, distributed shared memory architectures, this enables to enhance the locality of a mutual exclusion protocol by making each shared variable local to the process that accesses it most often.

One advantage of IMCs is that the *same* specification of a protocol can be used for both performance evaluation and functional verification [15]. Although mutual exclusion protocols serve traditionally as basic examples to illustrate the use of model checkers, it is not obvious to find an accurate description of their correctness properties in the action-based setting. We revisit these properties and specify them concisely using MCL [32], an extension of alternation-free modal μ-calculus with data-handling constructs and fairness operators accepted as input by the EVALUATOR 4.0 on-the-fly model checker. We observe that certain important properties are of linear-time nature, requiring formulas of $L\mu_2$ (the μ-calculus of alternation depth two) [12] or ACTL* [34]. Using MCL formulas parameterized by data values, we apply model checking also 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.

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. Finally, Section 4 gives some concluding remarks and directions for future work.

2 Background

After recalling 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 [2]. Concurrent processes communicate and synchronize only by means of atomic read/write 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). For simplicity, we consider in this study shared-memory protocols involving only two processes; as pointed out in [3], any mutual exclusion protocol for two processes can be generalized to $n \geq 2$ processes.

2.2 Modeling Mutual Exclusion Protocols using LOTOS NT

We specified the mutual exclusion protocols formally using LOTOS NT [6, 18], a variant of the E-LOTOS [23] 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 [5], 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 [3]. 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 and critical sections are modeled using the (non-synchronized) actions NCS and CS. The read/write operations on a shared variables 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 also for other protocols involving shared natural numbers.

As in LOTOS, emission and reception of values on a gate can take place simultaneously, as in the action "A (Read, 0, $?a_0$, j)" (where the values Read, 0, and j are emitted and a value is received in variable a_0), except that the variables holding the received values must be previously declared using a "**var**" statement. Unlike LOTOS, 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. To facilitate the

```
  loop
     non-critical section;
L0: A[j] := 0;
     if j = 1 and A[0] = 1 then
       goto L0
     end if;
     A[j] := 1;
     if j = 1 and A[0] = 1 then
       goto L0
     end if;
L1: if j = 0 and A[1] = 1 then
       goto L1
     end if;
     critical section;
     A[j] := 0
  end loop                                  (a)
```

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

```
process L [A:Operation, CS:Access, NCS:Pid, MU:Latency] is
  loop var ind, pid:Nat in select
      A (Read, ?ind, ?any Nat, ?pid); MU (Read, ind, pid)
      []
      A (Write, ?ind, ?any Nat, ?pid); MU (Write, ind, pid)
      [] ...
      CS (Enter, ?pid); MU (Enter, pid)
      []
      NCS (?pid); MU (Work, pid)
    end select end var end loop
end process                                 (e)
```

Fig. 1. Burns & Lynch protocol [5] for two processes: (a) Unstructured pseudo-code of process P_j ($j \in \{0, 1\}$); (b) LOTOS NT code of process P_j; (c) LOTOS NT code of the systems' architecture; (d) LOTOS NT code of the cell $A[ind]$ of the shared array; (e) LOTOS NT code of the auxiliary process L for inserting Markov delays.

specification of temporal properties (see Sec. 3.1), the critical section is split in two actions and each read/write operation carries the identifier of the underlying process. The LOTOS NT specification of process P_j 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. The additional process L (see Fig. 1(d)) serves to insert Markov delays at appropriate places in the model (see Sec. 2.3).

We specified 23 mutual exclusion protocols in LOTOS NT following the scheme shown in Figure 1: Burns & Lynch [5], Craig and Landin & Hagersten (CLH) [8, 28], Dekker [11], Dijkstra [10], Peterson [35], Knuth [26], Lamport [27], Kessels [25], Mellor-Crummey & Scott (MCS) [33], Szymanski [40], the black-white bakery protocol of [41], and twelve protocols generated automatically in [3]. Additionally, we also specified a trivial (incorrect) one-bit protocol for benchmarking purposes. The total size of the specifications (including comments, and after factoring common datatypes and processes in separate modules as much as possible) is about 2850 lines of LOTOS NT.

2.3 Transformation to Interactive Markov Chains

The LOTOS NT specification of each protocol is transformed into an Interactive Markov Chain (IMC) by adding Markov delays in a constraint-oriented style [15]. Precisely, we add a concurrent process L to the system consisting of the two processes and the shared variables. A skeleton of process L is shown in Figure 1(e).

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. The parameters of action MU allow to distinguish, for each process, between a read access, a write access, 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 obtained IMCs contain nondeterministic choices whenever two concurrent read/write 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 two actions (see Sec. 3.2 for details). This assumption is based on the fact that an uniform scheduler provides the best choice (in the sense of maximising entropy [38]) when no additional information is available about the choice of actions performed by the physical system. A more general solution, inspired by a technique used in the context of Markov decision processes [36], 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 [44], but is not yet available as an implementation).

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 [17]. The protocols were analyzed by model checking and performance evaluation, both kinds of analysis being automated using SVL [16] scripts.

3.1 Model Checking

We expressed the correctness properties of the mutual exclusion protocols as formulas in the MCL language [32], which extends the alternation-free μ-calculus [12] with regular expressions over transition sequences similar to those of PDL [13], data-handling constructs inspired from functional programming languages, and a (generalization of) the infinite looping operator of PDL-Δ [39]. MCL allows 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 [32], built using the OPEN/CÆSAR [14] graph exploration environment of CADP, implements an efficient on-the-fly model checking procedure for MCL, by translating MCL formulas into boolean equation systems and solving them on-the-fly using the algorithms of the CÆSAR_SOLVE library [31]. The model checker also exhibits full diagnostics (examples and counterexamples) as subgraphs of the LTS illustrating the truth value of MCL formulas.

MCL is roughly built from three kinds of formulas. First, *action formulas* A characterize 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, all variables assigned by value extraction being exported to the enclosing formula. Second, *regular formulas* R characterize 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^+"), and iterations bounded by counters ("$R\{n\}$"). Third, *state formulas* F characterize 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" ?j:Nat } . (not { CS !"LEAVE" !j })* .
  { CS !"ENTER" ?k:Nat where k <> j }
] false
```

This modality forbids the existence of sequences containing the entry of a process j in the critical section followed by the entry of another process $k \neq j$ in the critical section before process j has left its critical section. Note how the process index j is extracted from a transition label by the first action predicate "{ CS !"ENTER" ?j:Nat }" and is used subsequently in the formula.

Livelock freedom. This liveness property[1] 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 ?j:Nat } . (not { ?any ?"READ"|"WRITE" ... !j })* .
  { ?any ?"READ"|"WRITE" ... !j }
] mu X . (< true > true and [ not { CS !"ENTER" ?any } ] X)
```

The minimal fixed point formula binding the X variable expresses the inevitable execution of some critical section after process P_j 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 process can spin forever on reading shared variables. Figure 2(b) illustrates the counterexample of this formula exhibited by EVALUATOR 4.0 for Peterson's protocol. This protocol uses three shared variables, two of which being encoded as array cells A[0], A[1] and the third one by a separate variable B. The lasso-shaped diagnostic in Figure 2(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, a livelock situation occurs when both processes are executing cyclically 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 ?j:Nat } . (not { ?any ?"READ"|"WRITE" ... !j })* .
  { ?any ?"READ"|"WRITE" ... !j }
] not <  (not { CS ... })* . { ?G:String ... ?k:Nat where G <> "CS" } .
         (not { CS ... })* . { ?G:String ... !1 - k where G <> "CS" }
      > @
```

[1] Although some authors [3] use the term *deadlock* for this property, we prefer the term *livelock* used in [2]. Indeed, in the shared-memory setting involving only atomic read and write operations, the behavior of the system cannot contain deadlocks (i.e., sink states in the LTS), since each process can at any time execute some instruction.

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

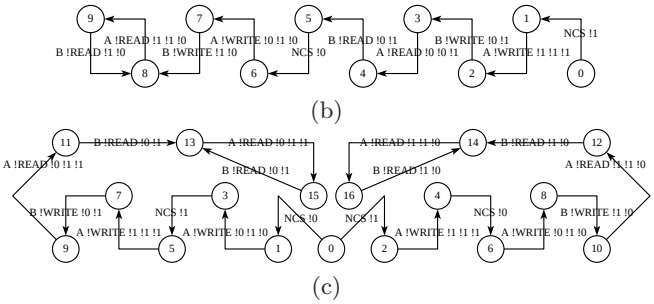


Fig. 2. (a) Peterson's protocol for process P_j ($k = 1 - j$); (b) Livelock produced by spinning of process P_0 when process P_1 "has decided to stop"; (c) Livelocks produced after P_0 or P_1 crashed while executing their entry sections.

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 the formula above, when translated to plain modal μ-calculus, belongs to the fragment $L\mu_2$ of alternation depth two [12], because the regular formula inside the infinite looping operator (which denotes a maximal fixed point) contains star operators (which denote minimal fixed points). Nevertheless, this formula is evaluated in linear-time by the algorithm proposed in [32], which generalizes the detection of accepting cycles in Büchi automata.

We can also observe that (a state-based version of) this formula cannot be specified in LTL [30], because it expresses the existence of sequences (denoted by the < ... > @ operator) starting from various states of the LTS (the states at the end of the subsequences captured by the [...] modality) and not only from the initial state of the LTS. However, as it was pointed out in [3], livelock freedom can be expressed just by forbidding the existence of unfair cycles (assuming that the initial state of the LTS can be reached from any other state, which holds for all protocols considered here). Therefore, the box modality can be dropped and the resulting formula can be expressed in LTL.

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 ?j:Nat } . (not { ?any ?"READ"|"WRITE" ... !j })* .
  { ?any ?"READ"|"WRITE" ... !j }
] not <  (not { CS ... !j })* . { ?G:String ... ?k:Nat where G <> "CS" or k <> j } .
         (not { CS ... !j })* . { ?G:String ... !1-k where G <> "CS" or 1-k <> j }
      > @
```

The < ... > @ operator describes a cycle containing at least one action performed by each process, but no entry of process P_j in its critical section. The formula belongs to $L\mu_2$, but (a state-based version of) it can also be expressed in LTL in the same way as livelock freedom.

Bounded overtaking. Even if a mutual exclusion protocol is starvation-free, it is interesting to know, when a process P_j begins its entry section, how many times the other process P_k can access its critical section before P_j 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 !0 } . (not { ?any ?"READ"|"WRITE" ... !0 })* .
  { ?any ?"READ"|"WRITE" ... !0 } .
  (  (not { CS ?any !0 })* . { ?G:String ... !0 where G <> "CS" } .
     (not { CS ?any !0 })* . { CS !"ENTER" !1 }
  ) { max }
> true
```

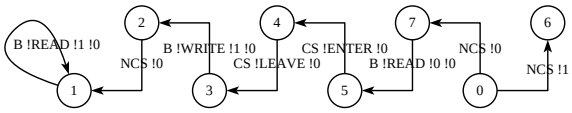
This formula expresses the existence of a sequence in which process P_0 executes its non-critical section, then the first instruction of its entry section, followed by *max* repetitions of a subsequence in which P_0 executes some instruction but only P_1 enters its critical section (a symmetric formula must be checked to determine the overtaking of process P_1 by P_0). For each starvation-free protocol, there exists a value of *max* such that the formula above holds for *max* and fails for $max + 1$. To minimize the number of model checking invocations, one can start with $max = 1$ and (if the formula holds for this value) keep doubling it until finding the first value max' for which the formula fails, then use a dichotomic search to reduce the size of the interval $[1, max']$ to 1.

Independent progress. A requirement formulated explicitly by Dijkstra [10] 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 often stated aside in the definition of the framework [5, 2]. However, we believe that this requirement is fundamental (at least from a model checking point of view), and should be verified separately. In MCL, it can be expressed using the following formula:

```
forall j:Nat among { 0 ... 1 } .
  [ true* ] (< { NCS !1-j } > true implies < { ... !j }* . { CS ... !j } > @)
```

which states that whenever the process P_k (where $k = 1 - j$) is about to enter its non-critical section, then the other process P_j can freely execute its code. Note that this formula belongs to $L\mu_2$ and can be also expressed in ACTL* but not in LTL, because it states the existence of infinite sequences starting from several (unknown) states of the LTS. As regards 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, whose model checking problem can be translated into the evaluation of a single < ... > @ operator that encodes the underlying Büchi automaton.

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



(b)

Fig. 3. (a) Trivial one-bit protocol for process P_j ($k = 1 - j$); (b) Counterexample for the independent progress of P_0 when P_1 has stopped in its non-critical section.

To see that the property of independent progress is not implied by the three other properties of mutual exclusion protocols, consider the trivial one-bit protocol shown in Figure 3(a). This simple 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 3(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 [5] (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 process may spin forever without entering its critical section. For all protocols considered here, we checked that this indeed holds; Figure 2(c) shows the diagnostic produced by EVALUATOR 4.0 illustrating, for Peterson's protocol, the livelock of each process when the other one has crashed after executing the first instruction of its entry section.

Model checking results. Table 1 summarizes the model checking results for the protocols considered. The generation of the IMCs for all protocols takes about 1 minute and a half on a standard desktop computer. Because the IMCs are small, the execution of the SVL script (48 lines) implementing the model checking of all properties on all protocols takes about 10 minutes. All properties have been checked on-the-fly using LNT.OPEN and EVALUATOR 4.0.

All the protocols considered satisfy the mutual exclusion, livelock freedom, and (except the trivial) the independent progress properties stated above. As regards the overtaking of processes, all starvation-free protocols (except Szymanski's) are symmetric, the minimal (1) and maximal (4) amount of overtaking being reached by Knuth's and by Dekker's protocol, respectively. The unbounded overtaking of one process by the other one has been checked by replacing, in the bounded overtaking formula given above, the bounded itera-

Protocol (2 processes)	Number of variables	IMC size		L/S-free	Overtaking	
		states	transitions		P_0/P_1	P_1/P_0
trivial	1	89	130	S	1	1
Burns & Lynch	2	259	368	L	∞	3
Szymanski		547	803	S	2	1
2b_p1		259	369	L	∞	1
2b_p2		271	386	L	∞	1
2b_p3		277	392	L	1	∞
Dekker	3	599	856	S	4	4
Knuth		917	1312	S	1	1
3b_p1		486	690	S	3	3
3b_p2		627	879	L	∞	1
Peterson		407	580	S	2	2
3b_c_p1		627	884	S	2	2
3b_c_p2		407	580	S	2	2
3b_c_p3		363	516	S	2	2
Lamport	4	1599	2274	L	∞	∞
Kessels		1073	1502	S	2	2
CLH		690	936	S	2	2
4b_p1		432	610	L	∞	1
4b_p2		871	1229	S	3	3
4b_c_p1		1106	1542	L	∞	1
4b_c_p2		1106	1542	L	1	∞
Dijkstra	5	899	1260	L	∞	∞
MCS		424	612	S	2	2
B&W Bakery	7	31222	43196	S	2	2

Table 1. Model checking results: the first column gives the name of the protocol; the second column gives the number of shared variables; the third and fourth columns give the size of the IMC; the fifth column indicates whether the protocol is only livelock- (L) or livelock- and starvation-free (S); the last two columns give the maximal number of times process P_j can overtake process P_k in accessing the critical section (P_j/P_k).

tion operator $R\{max\}$ by an infinite looping operator `<...>@`. All livelock-free, but not starvation-free protocols (except Dijkstra's and Lamport's) are asymmetric w.r.t. overtaking, only one process being able to overtake the other one 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. All delays being equal, the higher the throughput, the more efficient the protocol, because the longer a process is in the critical section, the less time it spends executing the protocol or waiting to enter the critical section.

Performance evaluation of an IMC is based on the transformation of the IMC into a Continuous-Time Markov Chain (CTMC) extended with probabilistic choices. 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), and (2) each MU action is transformed into an exponential delay by associating a rate λ to it, i.e., renaming it into"rate λ". Using exponential delays reflects 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.

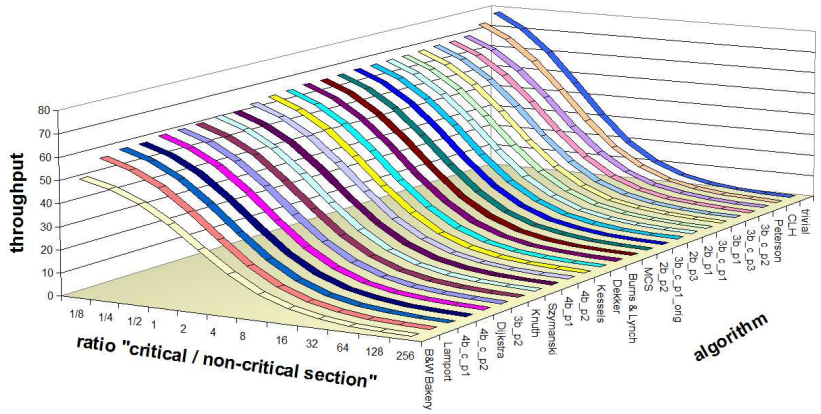
In all our experiments, we kept the rates for accesses to the shared variables constant: each read access has rate 3000 and each write access has rate 2000, reflecting that, on average, a write access is generally slower than a read access. For complex operations, namely *fetch-and-store* (used by the protocols CLH and MCS) and *compare-and-swap* (used by MCS), we used the same rate as for a write access. We also kept the rate for the critical section constant at 100, i.e., making the assumption that the critical section contains (on average) several read and write accesses. Hence, we varied only the delay for the non-critical section of both processes to compare the protocols in different usage scenarios.

In a second step, the stochastic LTS is minimized for stochastic branching bisimulation [22]. Unfortunately, this does not yield a CTMC, because due to the nondeterminism only some, but not all, of the i actions are eliminated. As discussed in Section 2.3, this nondeterminism is resolved by assuming an uniform scheduler. Practically, each nondeterministic choice is replaced by a uniform probabilistic choice, by renaming all i transitions into "prob 0.5".

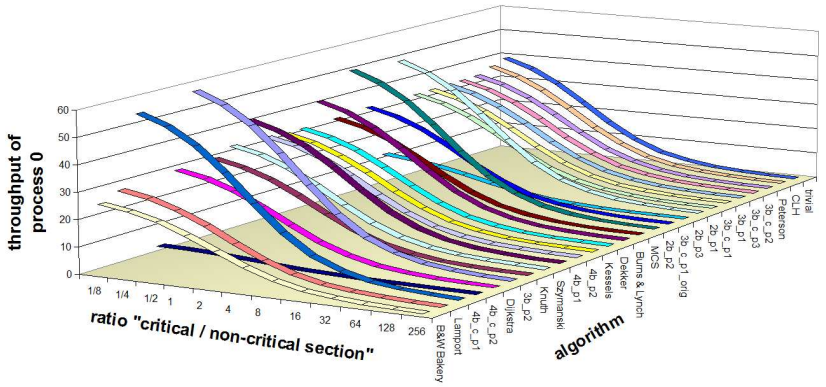
Finally, we compute the throughput of the entries into the critical sections by both processes in the steady state using the BCG_STEADY tool [20], which is able to handle CTMCs extended with probabilistic choices. The results of our experiments are shown in Figures 4 to 6. Because these figures depend on the arbitraily chosen rates, the concrete values are, although exact up to floating point errors, less interesting than the relations and tendencies.

The performance evaluation experiments are automated by an SVL script (160 lines); computing all shown performance measures requires less than ten minutes on a standard computer.

Figure 4 shows the effect of varying the ratio "critical-section-rate / non-critical-section-rate". Concerning the global throughput, the results should not be surprising. A first observation is that the longer the non-critical section with respect to the critical section (and the accesses to the shared variables), the less the performance of the protocols differs. Conversely, the largest performance differences of the protocols are observed if the critical section is longer than the non-critical section. A second observation is that the complexity of the protocol (number of shared variables and length of entry and exit sections) impacts its performance: the most complex protocol (B&W Bakery) is the least efficient, whereas the trivial one-bit protocol is the most efficient, the second most efficient being CLH, followed by Peterson's protocol.



(a) Global throughput



(b) Throughput of process 0

Fig. 4. Performance when varying the ratio critial-section-rate/non-critical-section-rate

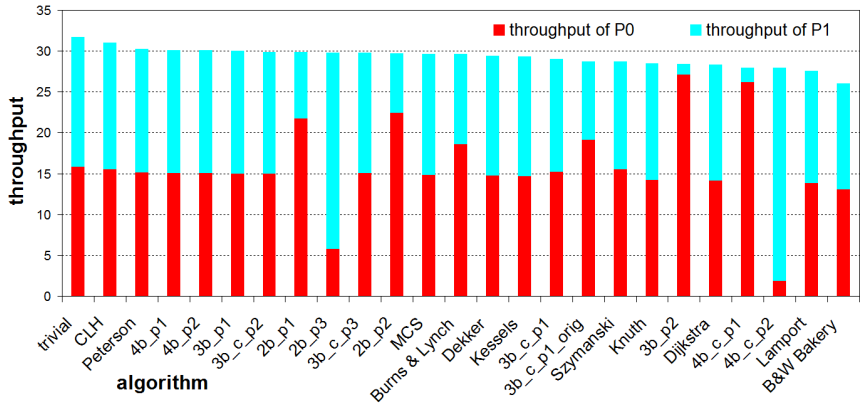


Fig. 5. Relative throughputs (ratio rate critical section/rate non-critical section = 2)

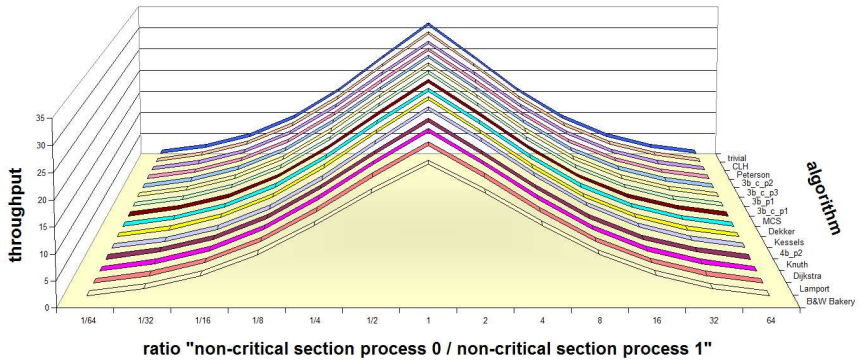
Concerning the throughput of process 0, the values are wider spread than for the global throughput. This difference is related to the symmetry concerning bounded overtaking of the protocols. For symmetric protocols, where the processes can overtake each other the same number of times, the throughput of process 0 is half the global throughput. For asymmetric protocols, the throughput of process 0 is either higher (if process 0 can overtake process 1 more often) or lower (if process 1 can overtake process 0 more often) than half the global throughput. Thus, the highest throughput for process 0 is obtained by some automatically generated asymmetric protocols (3b_p2 and 4b_c_p1).

Figure 5 shows the throughputs of all protocols, using 50 for the rate of the non-critical section (thus, the non-critical section is, on average, two times as long as the critical section). One observes significant differences in the throughputs of the two processes if and only if the protocol is asymmetric; 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.

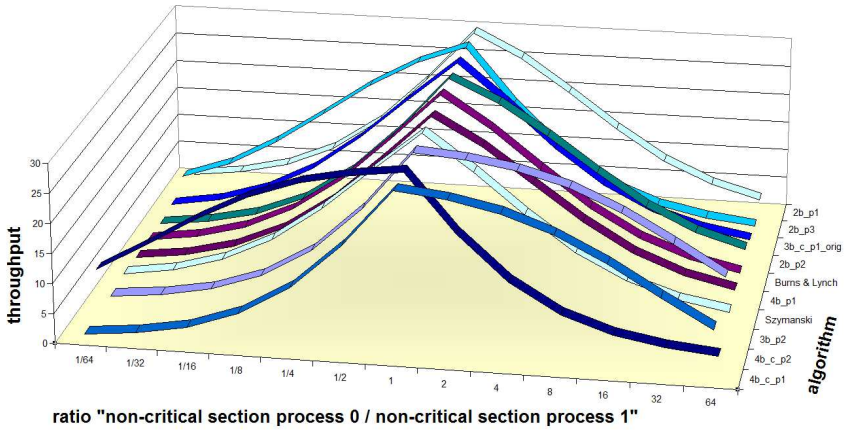
We also observed that making a protocol symmetric might (slightly) improve its performance. For instance, the original version of the automatically generated protocol 3b_c_p1 as described in [3] is asymmetric: with the same rates as in Figure 5, the throughput of process 0 (14.7499) is lower than the throughput of process 1 (15.0387). However, the symmetric version (that was used throughout this paper) has a higher global throughput of 29.8854 (instead of 29.7886, i.e., an increase of 0.3%, to be compared with the 20% performance improvement between the least and most efficient protocol).

The three plots of Figure 6 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 0 is slowed down (by decreasing the rate of the non-critical section of process 0); towards the right, process 1 is slowed down (by decreasing the rate of the non-critical section of process 1).

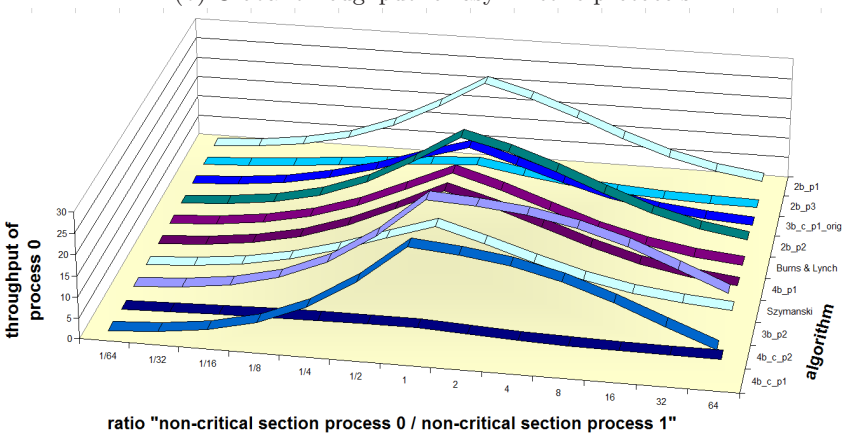
Figure 6(a) graphically justifies the name "symmetric" protocols: they are symmetric in the sense that slowing down process 0 has exactly the same effect on the global throughput as slowing down process 1: in both cases the general throughput decreases in the same way. Figure 6(b) shows that the situation 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 impact too much the advantaged process. Figure 6(c) confirms this intuition. On the one hand, for all those asymmetric protocols where process 0 can overtake process 1 infinitely, slowing down process 1 has less impact on the throughput of process 0 than slowing down process 0. On the other hand, for the two protocols 2b_p3 and 4b_c_p2, where process 0 can be overtaken infinitely by process 1, slowing down process 1 has more impact on the throughput of process 0 than slowing down process 0.



(a) Global throughput for symmetric protocols



(b) Global throughput for asymmetric protocols



(c) Throughput of process 0 for asymmetric protocols

Fig. 6. Performance when varying the ratio ncs-rate-p_0/ncs-rate-p_1

4 Conclusion and Future Work

This study aimed at assessing the applicability of model-based approaches for analyzing the functional behavior and the performance of shared-memory mutual exclusion protocols. As underlying semantic model, we used IMCs [19], which provide a uniform framework suitable both for model checking and performance evaluation. We carried out the analysis of 23 protocols using the state-of-the-art functionalities provided by the CADP toolbox [17]: formal specification using the LOTOS NT imperative-style process-algebraic language; description of functional properties using the MCL data-based temporal language; manipulation of IMCs by minimization and steady-state analysis using the BCG_MIN and BCG_STEADY tools; 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 [39], which can be checked in linear-time [32]. 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 [10]. 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.

An interesting future work direction is to continue the performance evaluation study for *adaptive* mutual exclusion protocols involving $n > 2$ processes, which so far were subject only to analytical studies [1]. Another direction would be a more detailed modeling of the underlying hardware architecture, in particular non-uniform memory access times. For instance, knowing which process accesses a variable most frequently might guide the placement of that variable to the local memory of the appropriate processor in the architecture.

References

1. J. H. Anderson and Y.-J. Kim. Adaptive mutual exclusion with local spinning. *Proc. of ISDC'00*, LNCS 1914, pp. 29–43, 2000.
2. J. H. Anderson, Y.-J. Kim, and T. Herman. Shared-memory mutual exclusion: major research trends since 1986. *Distributed Computing*, 16:75–110, 2003.
3. Y. Bar-David and G. Taubenfeld. Automatic discovery of mutual exclusion algorithms. *Proc. of DISC'03*, LNCS 2848, pp. 136–150, 2003.
4. M. Botincan. AsmL specification and verification of Lamport's bakery algorithm. *J. of Computing and Information Technology*, 13(4):313–319, 2005.

5. J. E. Burns and N. A. Lynch. Mutual exclusion using indivisible reads and writes. *Proc. of ACCCC'80*, pp. 833–842, 1980.
6. D. Champelovier, X. Clerc, H. Garavel, Y. Guerte, F. Lang, W. Serwe, and G. Smeding. Reference manual of the LOTOS NT to LOTOS translator (Version 5.0). INRIA/VASY, 107 pages, Mar. 2010.
7. G. Chehaibar, M. Zidouni, and R. Mateescu. Modeling multiprocessor cache protocol impact on MPI performance. *Proc. of QuEST'09*, IEEE Press, 2009.
8. T. S. Craig. Building FIFO and priority-queuing spin locks from atomic swap. Technical Report 93-02-02, University of Washington, Seattle, Feb. 1993.
9. G. Delzanno and A. Podelski. Model checking in CLP. *Proc. of TACAS'99*, LNCS 1579, pp. 223–239, 1999.
10. E. W. Dijkstra. Solution of a problem in concurrent programming control. *Commun. ACM*, 8(9):569–570, Sept. 1965.
11. E. W. Dijkstra. *Co-operating sequential processes*, pages 43–112. Academic Press, New York, 1968.
12. E. A. Emerson and C.-L. Lei. Efficient model checking in fragments of the propositional mu-calculus. *Proc. of LICS'86*, pp. 267–278, 1986.
13. M. J. Fischer and R. E. Ladner. Propositional dynamic logic of regular programs. *J. Comput. Syst. Sci.*, 18(2):194–211, Sept. 1979.
14. H. Garavel. Open/Cæsar: an open software architecture for verification, simulation, and testing. *Proc. of TACAS'98*, LNCS 1384, pp. 68–84, 1998.
15. H. Garavel and H. Hermanns. On combining functional verification and performance evaluation using CADP. *Proc. of FME'02*, LNCS 2391, pp. 410–429, 2002.
16. H. Garavel and F. Lang. SVL: a scripting language for compositional verification. *Proc. of FORTE'01*, pp. 377–392, 2001.
17. H. Garavel, F. Lang, R. Mateescu, and W. Serwe. CADP 2006: a toolbox for the construction and analysis of distributed processes. *Proc. of CAV'07*, LNCS 4590, pp. 158–163, 2007.
18. H. Garavel and M. Sighireanu. Towards a second generation of FDTs – rationale for the design of E-LOTOS. *Proc. of FMICS'98*, pp. 187–230, 1998.
19. H. Hermanns. *Interactive Markov chains and the quest for quantified quality*. LNCS 2428, 2002.
20. H. Hermanns and C. Joubert. A set of performance and dependability analysis components for CADP. *Proc. of TACAS'03*, LNCS 2619, pp. 425–430, 2003.
21. H. Hermanns and J.-P. Katoen. Performance evaluation:=(process algebra+model checking) Markov chains. *Proc. of CONCUR'01*, LNCS 2154, pp. 59–81, 2001.
22. H. Hermanns and M. Siegle. Bisimulation algorithms for stochastic process algebras and their BDD-based implementation. *Proc. of ARTS'99*, LNCS 1601, pp. 244–265, 1999.
23. ISO/IEC. Enhancements to LOTOS (E-LOTOS). International Standard 15437:2001, International Organization for Standardization, Genève, Sept. 2001.
24. H. E. Jensen and N. A. Lynch. A proof of Burns N-process mutual exclusion algorithm using abstraction. *Proc. of TACAS'98*, LNCS 1384, pp. 409–423, 1998.
25. J. L. W. Kessels. Arbitration without common modifiable variables. *Acta Informatica*, 17:135–141, 1982.
26. D. E. Knuth. Additional comments on a problem in concurrent programming control. *Commun. ACM*, 9(5):321–322, May 1966.
27. L. Lamport. A fast mutual exclusion algorithm. *ACM Transactions on Computer Systems*, 5(1):1–11, Feb. 1987.
28. P. S. Magnusson, A. Landin, and E. Hagersten. Queue locks on cache coherent multiprocessors. *Proc. of IPPS'94*, pp. 165–171, 1994.

29. Z. Manna and A. Pnueli. *Tools and rules for the practicing verifier*, pages 125–159. ACM Press and Addison-Wesley, 1991.
30. Z. Manna and A. Pnueli. *The temporal logic of reactive and concurrent systems*, volume I (specification). Springer Verlag, 1992.
31. R. Mateescu. CAESAR_SOLVE: a generic library for on-the-fly resolution of alternation-free boolean equation systems. *STTT*, 8(1):37–56, 2006.
32. R. Mateescu and D. Thivolle. A model checking language for concurrent value-passing systems. *Proc. of FM'08*, LNCS 5014, pp. 148–164, 2008.
33. J. M. Mellor-Crummey and M. L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21–65, Feb. 1991.
34. R. D. Nicola and F. W. Vaandrager. *Action versus state based logics for transition systems*, LNCS 469, pp. 407–419, 1990.
35. G. L. Peterson. Myths about the mutual exclusion problem. *IPL*, 12(3):115–116, 1981.
36. M. L. Puterman. *Markov decision processes: discrete stochastic dynamic programming*. Wiley, 1994.
37. M. Raynal. *Algorithmique du parallélisme : le problème de l'exclusion mutuelle*. Dunod-Informatique, Paris, 1984.
38. A. Shiryaev. *Probability*. Springer, 1996.
39. R. Streett. Propositional dynamic logic of looping and converse. *Information and Control*, (54):121–141, 1982.
40. B. K. Szymanski. A simple solution to Lamport's concurrent programming problem with linear wait. *Proc. of ICSS'88*, pp. 621–626, 1988.
41. G. Taubenfeld. The black-white bakery algorithm and related bounded-space, adaptive, local-spinning and FIFO algorithms. *Proc. of DISC'04*, LNCS 3274, pp. 56–70, 2004.
42. G. Taubenfeld. *Synchronization algorithms and concurrent programming*. Pearson, Prentice Hall, 2006.
43. J.-H. Yang and J. H. Anderson. A fast, scalable mutual exclusion algorithm. *Distributed Computing*, 9(1):51–60, Aug. 1995.
44. L. Zhang and M. R. Neuhäußer. Model checking interactive Markov chains. *Proc. of TACAS'10*, LNCS 6015, pp. 53–68, 2010.
45. X. Zhang, Y. Yan, and R. Castaneda. Evaluating and designing software mutual exclusion algorithms on shared-memory multiprocessors. *IEEE Parallel Distributed Technology*, 4(1):25–42, 1996.