Mutual Exclusion with Random Times

AVACS S3 Benchmark[⋆]

[1] Albert-Ludwigs-Universität Freiburg, Fahnenbergplatz, 79085 Freiburg, Germany
[2] Carl von Ossietzky Universität Oldenburg, 26111 Oldenburg, Germany
[3] Universität des Saarlandes, 66041 Saarbrücken, Germany

1 Introduction

In this benchmark we exemplify the effects of interactive Markov chain (IMC) modelling and model transformation to continuous-time Markov decision processes (CTMDPs) on a non-trivial example. We consider a solution to the mutual exclusion (mutex) problem that is based on the Fischer/Lamport algorithm [12], but enhanced with random times [3,4]. The basic idea of this variant is to equip the necessary mutex operations with random delays – governed by negative exponential distributions. This enables the study of performance aspects of the algorithm. We refer to the detailed exposition in [3,4] and recall here the basic definitions and notions used for describing mutual exclusion of processes, which we will here call processors.

We model the functional part of the algorithm in terms of Lotos [9] processes using the Cadp [6] toolbox. Including timed behaviour into the functional model is realised as exposed in [2]. Generating IMCs and ensuring the necessary property of being uniform is treated in more detail in [10,8]. In this case study we have made extensive use of the Svl scripting language [5] integrated in Cadp.

In the following we assume that the reader is familiar with the concepts of IMCs [7] and CTMDPs as, for instance, given in [13].

2 Terminology

Given that we refer to parts of Lotos specifications as *processes* we will not overload this term by reusing it for the participants of a mutual exclusion procedure. In the following a process in the scenario of mutual exclusion will be referred to as *processor*. An important concept in mutual exclusion problems is that of a *lock*. A lock comprises of a shared variable and three different operations on it. These are

– *inspecting*, where the lock is inspected whether it is empty, i. e., it is checked if its variable equals 0,
– *reading*, this operation reads the actual value stored in the locks' variable and returns it,
– *writing*, allows a write operation on the locks' variable.

Each of these operations is assumed to take a randomly distributed amount of time to finish. In our models we assume all distributions to be negative exponential, but with different parameters

[⋆] `http://www.avacs.org`

Algorithm 1 Fischer/Lamport mutual exclusion algorithm

```
x, y: shared variables, initially 0;
J: processor index;
                                                    ▷ remainder region
L:                                                  ▷ trying region
if x != 0 goto L;
x := J;

if x != J goto L;

if y != 0 goto L;
y := 1;

if x != J goto L;
enter critical section
                                                    ▷ critical region
exit critical section

y := 0;                                             ▷ exit region
x := 0;
endproc
```

for each of the three operations. A processor is said to *own a lock*, if it has written and read its unique identity number to and from it after inspecting the lock to be empty.

A *register* is a special kind of a lock whose shared variable is capable of storing exactly one bit. As a register can only store values 0 and 1, there is no functional distinction between inspecting and reading it.

We use this basic terminology to give a brief description of the different *regions* a processor can be in

- the *remainder region*, here a processor does all its work not related to its critical region,
- the *trying region*, where the entry of the critical region is coordinated, i. e., a processor tries to own (a) lock(s),
- the *critical region*, where exclusive access is required,
- the *exit region*, here a processor leaves its critical region and frees all locks owned by it which means that all locks are reset to 0.

3 Mutual exclusion algorithm

Algorithm 1 displays the Fischer/Lamport algorithm as described in [3]. The algorithm uses a lock `x` with an additional register `y`. If processor `J` has owned lock `x` it enters its critical region. Here, owning lock `x` depends on register `y`. Using the terminology of [3], the depicted algorithm has the following properties. The algorithm ensures mutual exclusion – at any time at most one processor is in its critical region. Furthermore, *weak deadlock-freedom* is ensured, i. e., if one particular processor is in its trying region and all other processors are concurrently in their remainder regions, then this particular processor will eventually enter its critical region. The

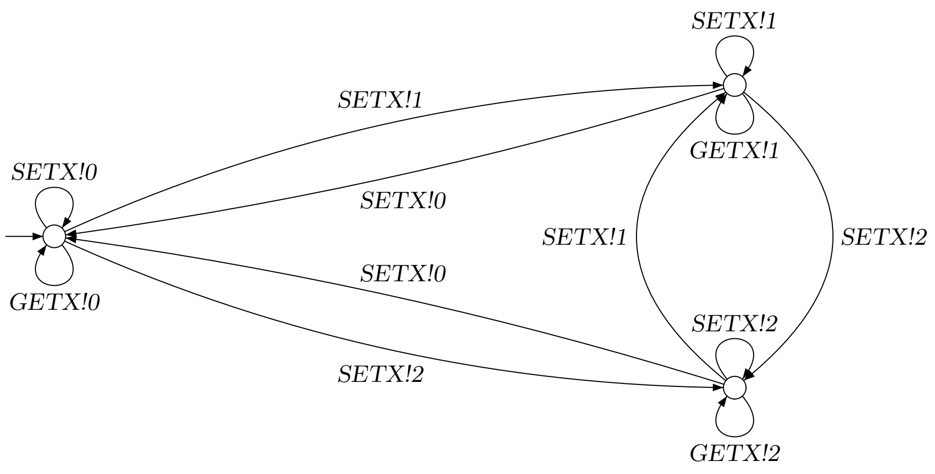

Fig. 1. Variable with range $\{0, 1, 2\}$.

same holds for the exit region. If exactly one processor is in its exit region and concurrently all other processors are in their remainder regions, all locks are eventually reset. The stronger version, called *deadlock-freedom*, is not ensured. In this setting deadlock-freedom requires that when some processor is in its trying (respectively, exit) region, subsequently some processor will be in its critical (remainder) region. However, this is not ensured by Algorithm 1. In fact, it is possible that the entire algorithm gets into a deadlock state.

4 Modelling with Cadp

For the mutual exclusion algorithm we have to model the two variables `x` and `y`, the processes and the time constraints of inspecting, reading and writing a variable.

In the following we show the principle structure of the LTSs representing the functional behaviour of the processors and of the variables, respectively. Together with the pictorial description we provide the belonging LOTOS processes.

Locks. In Figure 1 we show the LTS for variable `x` that has a range of $\{0, 1, 2\}$, i. e., two processors are considered. Actions *GETX!i*, for $i \in \{0, 1, 2\}$, are used to indicate that value i is read from variable `x`. Similar, actions *SETX!j*, for $j \in \{0, 1, 2\}$, are used to indicate that value j is written to variable `x`. Intuitively, in the initial state, the value of `x` is 0 and thus, only actions *GETX!0* and *SETX!j* for $j \in \{0, 1, 2\}$ are available. In particular, writing a certain value to `x` is always possible, but only the current value of `x` can be read. We have depicted the LTS just for two processors for the sake of readability and to show the general structure of LTSs representing variables. Variable `y` is modelled in exactly the same way, except that we use action names *GETY* and *SETY* instead of *GETX* and *SETX*, respectively. The range of variable `y` is $\{0, 1\}$ as it is a register comprising of a boolean variable.

The LOTOS processes for lock `x` and register `y` are shown in Appendix A as Process 1 and Process 2 on page 7.

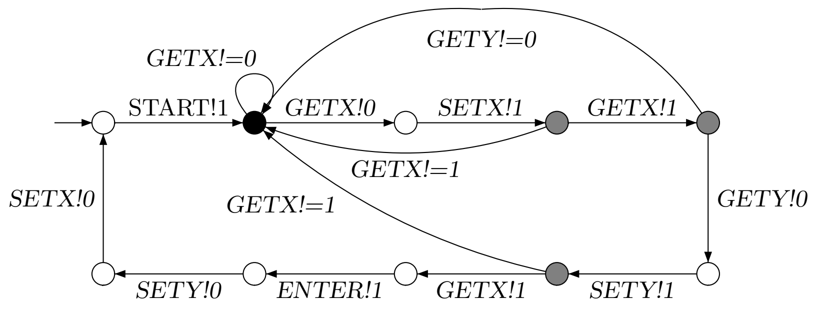

Fig. 2. General behaviour of processor with id 1

Processors. The LTS of a processor can directly be deduced from Algorithm 1. As an example, we show the general structure of an LTS representing the processor with unique id 1 in Figure 2. Here, we use actions *START!1* and *ENTER!1* to indicate that processor 1 enters its trying region and its critical section, respectively. Once the processor has entered its trying region it inspects lock `x`. Only if `x` is empty processor 1 continues. Otherwise, it remains in the black filled state until lock `x` is observed to equal 0. We have indicated this in Figure 2 by using action name *GETX!=0*. In all of the grey shaded states the processor behaves similarly: A variable has to be read (or inspected), and only if the value is as indicated by Algorithm 1 the processor is allowed to continue. Otherwise, it returns to the black filled state, representing the beginning of the trying region.

Now, all of the actions *GETX*, *GETY* and *SETX*, *SETY* have to be delayed. For example, in the black filled state, lock `x` is inspected. However, there are emanating actions *GETX!0* and *GETX!=0*, but by no means they are delayed in isolation, i. e., there is no delay for each of these actions. In fact, introducing a delay for each of these actions would introduce a race between Markov transitions, but there is only a singular delay covering the inspection in total. Therefore, we slightly modify the LTS shown in Figure 2 as follows. In the black state the delay for inspecting lock `x` has to be started. We do so by adding two transitions, as shown in Figure 3 (left). The action that has to be delayed equals *insp*, and the delay is started once action *sInsp* has occurred. The time constraint for *insp* is depicted on the right hand side of Figure 3.

The according Lotos processes (Process 3 to Process 7) modelling the processor with id `J` are shown in Appendix A.

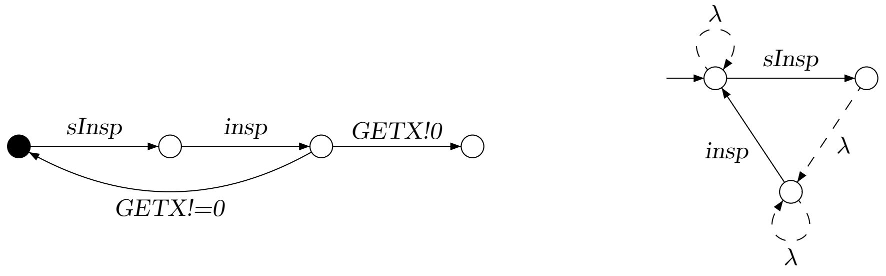

Fig. 3. LTS fragment and time constraint for inspection

#	Unminimised model States	Unminimised model Transitions	Generation time	Minimised model States	Minimised model Transitions	Minimisation time
1	17	28	1.45s	9	12	2.26s
2	797	2375	1.74s	149	422	2.30s
3	20883	83482	2.38s	1673	6301	3.45s
4	450209	22509191	26.08s	17943	87935	255.1s
5	8852319	53113612	1h 10m 33s	na	na	na

Table 1. System sizes and generation/minimisation times

5 Compositional generation

The uniform IMCs of the time constraints and the LTSs of the processors are composed as discussed in [10,8]. The uIMCs representing the individual processors are fully interleaved before synchronising with the variables. Here, the synchronisation set comprises of all read and write actions, i. e., $\{$*GETX!i, SETX!i, GETY!j, SETY!j*$\}$, where $i \in \{0, 1, \ldots, N\}$, N is the number of participating processors and $j \in \{0, 1\}$. Hiding of these actions in the composed model and a subsequent minimisation step yields the uIMC that we pass to the transformation [10]. However, the mutual exclusion example is not well scalable. In particular, for $N = 5$, i. e., for five participating processors, we were not able to minimise the model. Statistics of the generated model can be found in the next section.

6 Results

We have applied the transformation steps [10] to the IMCs obtained from the discussed specification for a different number of participating processors. Table 1 shows statistics on the unminimised and minimised models. Columns two and three show the number of states and transitions of the unminimised model, respectively. In column four we have depicted the generation time. The number of states and transitions of the minimised model and the minimisation times are shown in columns five, six and seven respectively. As mentioned above, we were not able to minimise the model with $N = 5$ processors.

A *strictly alternating* IMC directly corresponds to a CTMDP, and we show in Table 2 the sizes of the *strictly alternating* IMCs and the memory usage of the so-called underlying CTMDPs together with the transformation time. These statistics have been generated by our implementation of the transformation procedure from IMCs to CTMDPs. For a detailed explanation we refer to [10].

The computation of quantitative timed-reachability properties is carried out on the *underlying* CTMDP of the IMC. The proposed algorithm [1] for this has been integrated in an (yet) unofficial

#	# States Inter.	# States Markov	# Transitions Inter.	# Transitions Markov	Mem	Transf. time (s)
1	9	8	9	14	228 B	0.13
2	125	98	126	273	4.8 KB	0.14
3	1190	931	1195	3441	68 KB	0.16
4	10870	8418	10893	40876	849 KB	0.52

Table 2. Resulting CTMDP and transformation times

#	1 ms	2 ms	25 ms	50 ms	100 ms	
1	0.05	0.05	0.05	0.06	0.08	Runtime (s)
	144	153	168	193	243	Iterations
	0	0	0	0	0	Prob.
2	0.12	0.14	0.2	0.21	0.3	Runtime (s)
	145	163	193	243	343	Iterations
	0.00	0.17	0.35	0.56	0.81	Prob.
3	0.21	0.25	0.31	0.43	0.65	Runtime (s)
	146	173	218	293	443	Iterations
	0.00	0.26	0.5	0.74	0.93	Prob.
4	4.32	5.48	7.65	10.4	15.71	Runtime (s)
	147	183	243	343	543	Iterations
	0.00	0.25	0.54	0.8	0.96	Prob.

Table 3. Timed reachability analysis of the mutual exclusion algorithm

release of the MRMC [11] model checker. We have computed the worst-case probability to reach a deadlock state within t time units for different time bounds t. For this, the mean durations of exponential distributions for inspecting, reading and writing a variable are chosen as $1ms$, $2ms$ and $1.4ms$, respectively. We show the results in Table 3. For example, the second column shows the results for $t = 1ms$. In rows two to four the results for one participating processor are shown. The first number denotes the computation time, the second number equals the number of iterations and the third number shows the probability to reach a deadlock state. For one participating processor, the worst-case probability to reach a deadlock state always equals 0.

References

1. C. Baier, B. R. Haverkort, H. Hermanns, and J.-P. Katoen. Efficient Computation of Time-Bounded Reachability Probabilities in Uniform Continuous-Time Markov Decision Processes. *TCS: Journal on Theoretical Computer Science*, 345(1):2–26, 2005.
2. E. Böde, M. Herbstritt, H. Hermanns S. Johr, T. Peikenkamp, R. Pulungan R. Wimmer, and B. Becker. Compositional Performability Evaluation for Statemate. In *QEST: Conference on Quantitative Evaluation of SysTems*, pages 167–176. IEEE Computer Society, 2006.
3. E. Gafni and M. Mitzenmacher. Analysis of Timing-Based Mutual Exclusion with Random Times. In *Symposium on Principles of Distributed Computing*, pages 13–21, 1999.
4. E. Gafni and M. Mitzenmacher. Analysis of Timing-Based Mutual Exclusion with Random Times. *SIAM Journal on Computing*, 31(3):816–837, 2001.
5. H. Garavel and F. Lang. SVL: A Scripting Language for Compositional Verification. In *FORTE: Workshop on Formal Techniques for Networked and Distributed Systems*, pages 377–394, 2001.
6. H. Garavel, F. Lang, and R. Mateescu. An Overview of CADP 2001. In *EASST Newsletter*, volume 4, August 2002.
7. H. Hermanns. *Interactive Markov Chains and the Quest for Quantified Quality*, volume 2428 of *LNCS*. Springer, 2002.
8. H. Hermanns and S. Johr. Uniformity by Construction in the Analysis of Nondeterministic Stochastic Systems. In *DSN: International Conference on Dependable Systems and Networks*, pages 718–728, 2007.
9. ISO. *IS8807 : Information Processing Systems - Open System Interconnection - LOTOS - A formal description technique based on the temporal ordering of observational behavior.* ISO, February 1989.
10. S. Johr. *Model Checking Compositional Markov Systems.* PhD thesis, Universität des Saarlandes, 2007. submitted.

11. J.-P. Katoen, M. Khattri, and I. S. Zapreev. A Markov reward model checker. In *QEST: Conference on Quantitative Evaluation of SysTems*, pages 243–244. IEEE Computer Society, 2005.
12. N. Lynch and N. Shavit. Timing Based Mutual Exclusion. In *RTSS: Real-Time Systems Symposium*, pages 2–11, 1992.
13. M. L. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming.* Wiley, 1994.

A Lotos processes

In the following LOTOS processes we use

$$\mathcal{G} = \texttt{STARTME, sInsp, insp, sRead, read, sWrite, write, GETX, SETX,}$$
$$\texttt{GETY, SETY, ENTER}$$

as an abbreviation.

Process 1 Lock

```
process X [GETX, SETX] (VAL : Nat) : noexit :=
  GETX ! VAL; X [GETX, SETX] (VAL)
  []
  SETX ? x : Nat; X [GETX, SETX] (x)
endproc
```

Process 2 Register

```
process Y [GETY, SETY] (VAL : BOOL) : noexit :=
  GETY ! VAL; Y [GETY, SETY] (VAL)
  []
  SETY ? y : BOOL; Y [GETY, SETY] (y)
endproc
```

Process 3 Remainder region

```
process Remainder [$\mathcal{G}$] (J : Nat) : noexit :=
  STARTME ! J; TryOne [$\mathcal{G}$] (J)
endproc
```

Process 4 First part of trying region

```
process tryOne [𝒢] (J: Nat) : noexit :=
  sInsp; insp; GETX ? x : Nat;
  (
    [x == 0] → sWrite; write; SETX ! J; tryTwo [𝒢] (J)
    []
    [x <> 0] → tryOne [𝒢] (J)
  )
endproc
```

Process 5 Second part of trying region

```
process tryTwo [𝒢] (J: Nat) : noexit :=
  sRead; read; GETX ? x : Nat;
  (
    [x == J] → tryThree [𝒢] (J)
    []
    [x <> J] → tryOne [𝒢] (J)
  )
endproc
```

Process 6 Third part of trying region

```
process tryThree [𝒢] (J : Nat) : noexit :=
  sInsp; insp; GETY ? y : BOOL;
  (
    [NOT(y)] → sWrite; write; SETY ! TRUE; crit [𝒢] (J)
    []
    [y] → tryOne [𝒢] (J)
  )
endproc
```

Process 7 Last part of trying region, critical region and exit region

```
process crit [𝒢] (J : Nat) : noexit :=
  sRead; read; GETX ? x : Nat;
  (
    [x == J] → ENTER ! J;                                ▷ in critical section
    sWrite; write; SETY ! FALSE;                               ▷ exit region
    sWrite; write; SETX ! 0;
    remainder [𝒢] (J)
    []
    [x <> J] → tryOne [𝒢] (J)
  )
endproc
```