Journal of Distributed Computing and Systems(JDCS)
Vol.6, Issue 2, 2024, Pages 1-11

Modeling and Formal Verification of a Distributed Mutual Exclusion Algorithm

Leila NamvariTazehkand[1], Saied Pashazadeh[2] *

[1]*Department of Computer Engineering, Faculty of Electrical and Computer Engineering, University of Tabriz, Tabriz, East Azerbaijan, Iran.*

[2]*Department of Information Technology, Faculty of Electrical and Computer Engineering, University of Tabriz, Tabriz, East Azerbaijan, Iran.*

Article History:
Received: 22 November 2023
Accepted: 15 March 2024
Available online: 19 March 2024

Abstract

This paper presents the modeling and verification of a fully anonymous Mutual Exclusion (ME) algorithm, which implies that processes and memory are indistinguishable. The study utilizes Colored Petri Nets (CPN) to model the ME algorithm hierarchically, comprising low and top levels. Analysis of the state space diagram demonstrates that typically, only one process enters the critical section while others wait. Additionally, the study shows that different processes identify an arbitrary in-memory register with distinct identifiers. However, it reveals a potential deadlock scenario where two processes simultaneously acquire an equal number of registers and fail to release them, leading to deadlock. This paper presents the first modeling and verification of an ME algorithm using state-space analysis, highlighting the novelty of employing colored Petri nets. The presented model, encapsulating the essential property of full anonymity, can serve as a foundation for modeling similar distributed algorithms, thus establishing its significance as a reference framework in this domain.

Keywords: Formal Methods, Modeling, Verifying, Colored Petri Nets, Distributed Systems, Mutual Exclusion.

I. Introduction

This paper focuses on the modeling and verifying an ME algorithm [1]. This algorithm is fundamental in developing modern distributed systems because it is anonymous. Fully anonymous means that anonymous processes perform read/modify/write actions on anonymous registers. Process anonymity means that processes have no identity, and it is impossible to distinguish one process from another; thus, each process has its unique identifier but is unknown to other processes. Memory anonymity means that in a shared memory, a shared register *R1* by process *p* and a shared register *R2* by process *q* correspond to a shared register *R3* by process *X*, and the same register for all processes with identifiers can be identified differently.

In most existing research that has modeled mutual exclusion algorithms for verification, the processes know each other. Also, all the participating processes have a common view of the ID of the registers belonging to the memory. For example, Suryavanshi *et al.* [2] have modeled and specified Lamport's mutual exclusion algorithm for distributed systems using Event-B, where processes and memory are specified. Mateescu *et al.* [3] have modeled, and performance evaluated the mutual exclusion protocol using interactive Markov chains, where both processes and memory are specified. Also, Neilsen [4] has modeled and verified an ME algorithm using a real-time verification tool, *UPPAAL* (*Uppsala University and Aalborg University*), whose processes and memory are known for all participating processes. In addition, other ME algorithms have been modeled and verified [5-6], and none of the modeled algorithms are entirely anonymous.

In this paper, we model and specify the ME algorithm presented by Raynal *et al.* [1], aim to verify its properties and investigate the deadlock by one of the formal methods called Colored Petri Nets and the CPN tool. We model the algorithm hierarchically and in two levels called top and low levels; thus, the top level designs the competition of the participating processes to acquire the registries, and the low level designs the operation that each process performs alone according to the algorithm [1]. Then, we generate the state space graph with the CPN tool and check different algorithm scenarios using predefined and our developed CPN query functions.

We prove that in each step of the execution of the model, only one process enters the critical section; the other processes release the previously obtained registers and wait for another opportunity to enter the critical section. Thus, we show that if a process is in a lower round than other processes, it releases all already acquired registers. By contrast, if a process is in a higher round than other processes, it owns all the desired registers, enters its critical section, and finally releases all the registers it has previously acquired after its operation. Also, we prove that if two processes own an equal number of registers and are similar in a round, where *round* ≥ 2, then none of the two processes in question release the previously owned registers, and both will wait to acquire the rest of the registers, and this causes a deadlock in the algorithm. In addition, we prove that different processes recognize an arbitrary register with different identifiers, and

* Corresponding Author: Saied Pashazadeh (pashazadeh@tabrizu.ac.ir)

our model satisfies the fully anonymous property of the ME algorithm.

The paper's organization is as follows: Section II introduces the ME algorithm and then introduces colored Petri nets. Section III reviews related works. Section IV models the ME algorithm with colored Petri nets hierarchically in two levels. In Section V, the state space of the model is analyzed, and then the properties of the ME algorithm are proved. Section VI discusses the conclusion and future works.

II. Preliminaries

In this section, we introduce the fully anonymous mutual exclusion algorithms and the model checker colored Petri nets. Section A briefly introduces the ME algorithm, and Section B presents CPN.

A. A Summary of the Fully Anonymous Mutual Exclusion Algorithm

Raynal *et al.* [1] present the ME algorithm, which is executed in a system called *S*. They assume that the system *S* consists of a finite set of $n \geq 2$ asynchronous processes, denoted by $P_i = \{P_1, P_2, ..., P_n\}$. The system also includes a shared memory with $m>1$ unknown registers, where $R = [1,...,m]$ represents the registers. The process P_i only knows that the total number of processes is n and the number of registers is m and has no additional information about them. ME is introduced with a sequence of operations, which include: *acquire*(); *critical section; release*(); Thus, the critical section is the sequence of codes that P_i executes. P_i acquires the registers by invoking the function *acquire*() and enters the critical section, eventually releasing the obtained registers by invoking the function *release*(). Thus, the ME algorithm must satisfy the mutual exclusion property, meaning that no two processes can simultaneously be in the critical section. The process P_i has the local variables that include [1]: 1) max_i stores the maximal value contained in a register, as seen by P_i. 2) $counter_i$ stores the number of registers owned by P_i.

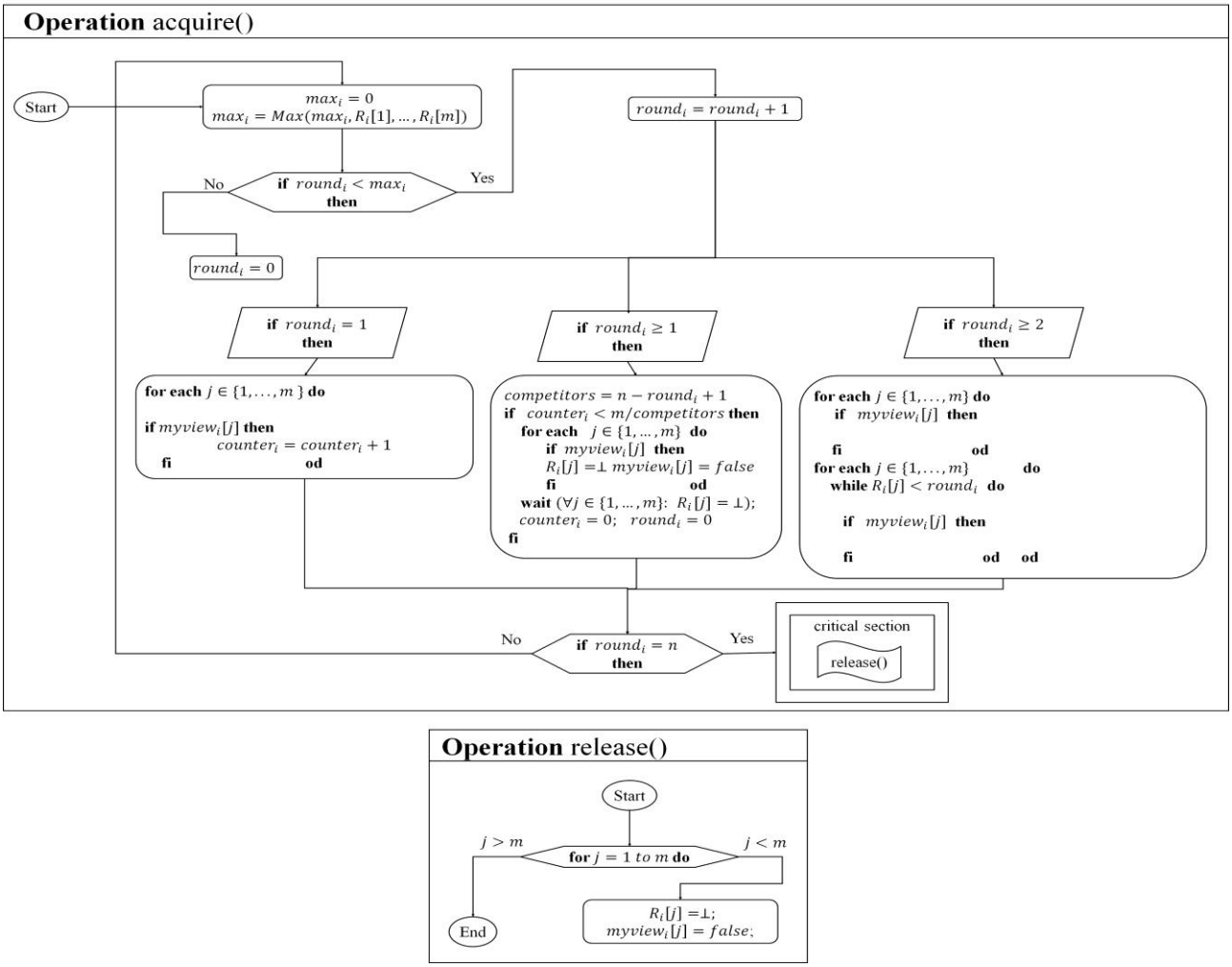

Fig. 1. The extracted diagram from the ME algorithm (for P_i)

A process owns a register when it is the last process that wrote a non-⊥ value into this register. 3) $myview_i[1..n]$ is an array of Boolean values, each initialized to false. When $myview_i[j]$ equals true, P_i acquires the register $R_i[j]$. 4) $round_i$ (initialized to 0) is the round currently attained by P_i in its competition to access the critical section. When $round_i = n$, P_i is the winner and can enter the critical section. Figure 1 illustrates an extracted diagram from the ME algorithm [1]. According to Fig. 1, P_i enters a repeat loop after starting and stays in that loop until $round_i$ equals n. When $round_i = r > 0$, P_i is in round r. P_i acquires more registers by writing the number r in the already owned registers. At the start of execution, P_i checks the contents of all registers and stores the maximum value in max_i. Then it compares the value of max_i and $round_i$; if no register has a ⊥ value, roundi < maxi and Pi stays at step 0, and roundi equals *0.* This means that other processes already own the registers and have a higher round value, and P_i remains at $round_i = 0$. Otherwise, P_i increases the value of $round_i$, and depending on the value of $round_i$, which may be $round_i = 1$, $round_i \geq 1$, and or $round_i \geq 2$, then chooses one of three paths in the diagram. Eventually, when P_i owns all registers, and $round_i$, equals n, P_i enters the critical section. Finally, P_i invokes *release()* from within the critical section, which resets all registers to their initial value and updates $myview_i[1,\ldots, m]$ to false.

B. Introduction to Colored Petri Nets

In 1962, Carl Adam Petri [7] introduced the Petri net. Petri net is a formal method based on bag theory for modeling, verifying, and validating behavioral properties in concurrent and distributed systems [7]. Petri net uses an extended graphical interface for modeling. The four main concepts of Petri net are *place*, *transition*, *token*, and *arc*. A place represents a system's possible states. An ellipse represents a place, and a place has tokens. A rectangle represents a transition. The transitions represent a system's actions or events that change its state. Transition can fire when it is enabled via the satisfaction of conditions mainly represented on tokens and inscriptions of input arcs or guard conditions of transition. Arcs connect a transition and place. A type of variable called color set is assigned to each place, whose data values are tokens of that type of color set, and the arcs connected to the place can only transfer a value compatible with the color set. Considering a transition, edges are input arcs if these connect a place to the transition, and output arcs connect the transition to a place. A type of variable called color set is assigned to each place, whose data values are tokens of that type of color set, and the arcs connected to the place can only transfer a value compatible with the color set [7].

The Colored Petri net is an extension of the classical Petri net, which arcs inscriptions, guard's conditions, and initial markings of places can be codded by Standard Machine Language (SML). CPN tool is a powerful tool that uses the concept of colored Petri net [8].

III. RELATED WORKS

This section introduces the presented ME algorithms in former years and reviews the techniques used to verify such algorithms.

Mutual Exclusion Algorithms: Raynal *et al.* [1] have introduced fully anonymous systems; an anonymous system means that the shared memory is also anonymous in addition to the processes. In other words, the processes do not agree on the name of the same registers, and a register has different names for different processes. Then, they present a deadlock-free mutual exclusion algorithm for fully anonymous systems where the anonymous registers are read/modify/write registers. Raynal *et al.* [9] presented a historical overview of Mutual Exclusion algorithms; they represented nine different algorithms developed between 1965 and 2020 and explained their features and design principles. Also, they have investigated those algorithms from the view of fully anonymous, anonymous memory and anonymous processes. Dhoked *et al.* [10] have presented an optimal solution to solve the Recoverable Mutual Exclusion (RME) problem; RME is a fault-tolerant variation of Dijkstra's classical ME in which a process can fail and recover. Daymude *et al.* [11] have proposed and rigorously analyzed a local mutual exclusion algorithm for nodes that are anonymous and communicate with other anonymous nodes via asynchronous message passing. Imbs *et al.* [6] have generalized the election problem for fully anonymous systems from two points of view. The first is d-election, in which at least one and at most d processes are elected. The second one is exact *d-election* in which exactly *d* (different) processes are elected. In their system model, processes and shared memory were anonymous and communicated through shared registers, where $d \in \{1,\ldots,n-1\}$. The presented mutual exclusion algorithms can be classified into permission-based and token-based [13-14].

In permission-based algorithms, a node enters its critical section only after receiving permission from a set of nodes; however, in token-based algorithms, a node enters its critical section when given a unique token. Neilsen [15] has presented a prioritized token-based algorithm for distributed mutual exclusion, which imposes very little storage overhead in each message and on each node. Mostéfaoui *et al.* [16] have presented a new algorithm implementing an array of n SWMR (single-writer/multi-reader) atomic registers in a peer-to-peer asynchronous message-passing system where each register is placed in a process and up to $t < n/3$ processes may commit Byzantine failures. *Related works about formal verification:* Mateescu *et al.* [3] have model checking and performance evaluated the mutual exclusion protocol using interactive Markov chains and with the Cadp toolbox. Suryavanshi *et al.* [2] have modeled and specified Lamport's mutual exclusion algorithm for distributed systems using Event-B; they have considered a vector clock instead of using Lam-port's scalar clock for message time stamping. Baarir *et al.* [17] have presented formal modeling and the verification of a new generic hierarchical approach.

This approach is based on the combination of classical distributed mutual exclusion algorithms that have already been proven correct. Moreover, they prove that their compositional approach preserves the properties of the mutual exclusion paradigm. Cicirelli *et al.* [5] have presented an approach to modelling and verification of mutual exclusion algorithms. Their approach is based on Timed Automata and is used from the *UPPAAL* (Uppsala University and Aalborg University) toolbox. They have also proposed a mutual exclusion algorithm for $N \geq 2$ processes whose model checking confirms all its properties. Ogata *et al.* [18] have presented an approach to *Ricart&Agrawala* distributed mutual exclusion algorithm. They have modeled the algorithm as a *UNITY* computational model and described it in *CafeOBJ*, where CafeOBJ is a language for writing a formal specification of the model. Also, they verified that the algorithm is mutually exclusive based on the CafeOBJ document. If up to k processes can enter their critical sections, this is called k-exclusion [19]. Zhao et al. [19] have presented token-based models that verify and validate k-mutual exclusion and m-mutual inclusion algorithms, where k denotes the maximum number of processes in their critical sections, and m represents the minimum number that must remain in their critical sections. Neilsen [4] has presented the models to analyze the performance and verify the correctness of the generalized algorithm. He has used a real-time verification tool, UPPAAL [20], and verified safety and liveness properties.

Reviewing related works, we observe that a fully anonymous mutual exclusion algorithm has not been modeled and verified using formal and mechanical methods. Also, until now, colored Petri nets have not been employed to model and verify an ME algorithm. Colored Petri nets offer the unique capability, compared to other tools, to stepwise show the operations performed by an anonymous process in obtaining anonymous registers within the state space analysis.

IV. MODELING THE ME ALGORITHM USING CPN

This section presents the CPN model of the ME algorithm. Section A defines the data structures used in the model, and Section B and Section C describe the top and low levels of the model, respectively. Since ME is a symmetric distributed algorithm, all participating processes execute the same code. Therefore, to have a lightweight model and a full state space graph, we consider the number of processes participating in the model to be the minimum number of three processes. We model the ME algorithm hierarchically at the top and low levels, where the top level introduces the competition of the participating processes to obtain the registries. The low level introduces the operation that each process performs alone according to the algorithm [1].

A. Data Structures

To specify the model, we first define data types called colset and then define variables of their type, as well as constants as initial marking. TABLE. I illustrate all the color sets used in the model. TABLE. II illustrates the variables used in the model. We define constants as the initial marking for places *P1*, *P2*, *P3*, and *Memory*, corresponding to processes named *process1*, *process2*, *process3*, and *Memory*, respectively:

- process1={P="P1",Myview=[false,false,false,false], Counter=0, Round=0, Max=0}
- process2={P="P2", Myview=[false,false,false,false], Counter=0, Round=0, Max=0}
- process3={P="P3",Myview=[false,false,false,false], Counter=0, Round=0, Max=0}
- Memory={Registers=[~1,~1,~1,~1],Withdrawal=false, Critical_section= ""}

In Section B and Section C, we describe the use of each data structure introduced in this section.

TABLE I. THE COLOR SETS

colset counter=int;
colset round=int;
colset withdrawal=bool;
colset Regs=list INT;
colset myView = list BOOL;
colset Cs=STRING;
colset memory = record Registers:Regs*Withdrawal: withdrawal*Critical_section:Cs;
colset process=record P:STRING*Myview:myView *Counter:counter*Round:round*Max:max;
colset C_S = product INT*BOOL;
colset R_M = product INT*INT;
colset my_reg=product myView*Regs;
colset Rand= list INT;

TABLE II. THE VARIABLES

var prcs:process;
var mem:memory;
var a:Rand;
var r:INT;

B. The Top-Level

This section describes how to implement the top level. Figure 2 illustrates the module called the main page. P1, P2, and P3 places represent the state of the processes P_1, P_2, and P_3, respectively. These places have a color set of the type *process*. The color set *process* is a record with five elements that represent, in order, the *process ID*, *myview, counter, round*, and *max*. The places *P1*, *P2*, and *P3* have an initial marking defined by the constant *process1, process2*, and *process3,* respectively. For example, *process1* is equal to *{P="P1", Myview=[false,false,false,false], Counter=0, Round=0, Max=0}.* The transitions called *release1*, *release2*, and *release3* represent operations P_1, P_2, and P_3 aimed at releasing registers previously obtained.

We also define three places called *flagp1*, *flagp2*, and *flagp3*, which are flags for releasing the registers by P_1, P_2, and P_3, respectively. For example, consider *flagp1*, whose initial marking is an empty list. When process P_1 owns all

Journal of Distributed Computing and Systems(JDCS)
Vol.6, Issue 2, 2024, Pages 1-11

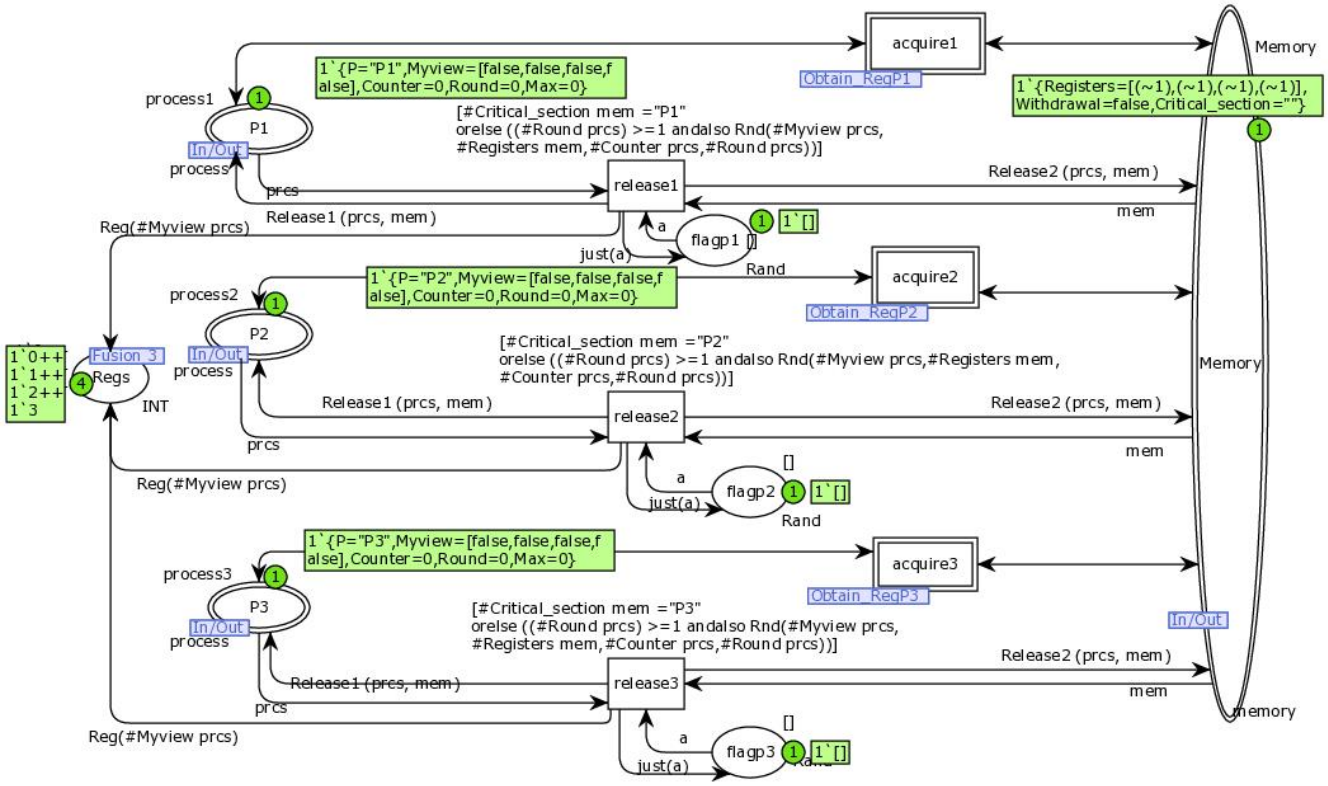

Fig. 2. The CPN model of the top level

registers and enters the critical section, the condition for releasing the registers owned by process P_1 is met; the condition is *Round* ≥ *1* and *Counter* < *m/competitors* where *competitors* = *n* - *Round* + *1*. Thus, flagp1 contains a token in its list, which causes the transition *release1* to be enabled and fired. By firing transition *release1*, all the registers are added to place *Regs*. Also, the values of the round, counter, and myview variables in place P1 and the values of the registers in place memory are changed to their initial values, and the process P_1 again participates in the competition to own the registers. Releasing the registers is also valid for processes P_2 and P_3.

We define three substitution transitions called *acquire1, acquire2*, and *acquire3* to replace the low-level and perform register acquisition operations by the processes P_1, P_2, and P_3, respectively. In addition, we define a place called *Regs* that represents the status of registers, where it has a color set of the type *INT*, and its initial markings indicate four registers with numbers *0, 1, 2*, and *3*. Moreover, we define a place called *memory*, which represents the state of shared memory. The place *memory* has a color set of the type *memory* and the initial marking of the constant *memory*, which the constant value of *memory* is equal to {*Registers* = [*~1,~1,~1,~1*], *Withdrawal=false, Critical_section= ""*}; Where the element *Registers* denotes the status of the registers, indicating whether a process has acquired a specific register. The *Withdrawal* and *Critical_section* introduce the status of memory; Thus, if a process enters the critical section, the value of *Withdrawal* changes from false to true, and the value of *Critical_section* is equal to the ID of the process that entered the critical section.

In Fig. 2, if the place *Regs* contains a token (say register), one of the substitution transitions *acquire1*, *acquire2*, or *acquire3* (e.g., *acquire1*) is enabled and fired. Assume transition *acquire1* fires; thus, the token is removed from the place *Regs*, and in the low level related to the process P_1, P_1 performs the operations associated with owning the register removed from the place Regs, which we describe in *S*ection C. According to Fig. 2, none of the three places, P_1, P_2, and P_3, are related to each other, and an arbitrary place among these three places indicates a process (e.g., P_1) has no interference in obtaining and releasing registers by another process (e.g., P_2, or P_3). Thus, the condition of anonymous processes is satisfied.

C. The Low-Level

This section models a process's operation to own registers at a low level. Three substitution transitions we have defined at the top level in Fig. 1 are used as submodules at the low level. Figures 3, 4, and *5* illustrate the submodules of *Obtain_RegP1*, *Obtain_RegP2* and *Obtain_RegP3*, respectively. In Fig. 3, we use the places P_1 and *Memory* as port/sockets in the module of the top level. It makes the place *P1* (respectively, the place *Memory*) transfer information related to the process P_1 (respectively, the shared memory) from the top level to the low level and vice versa.

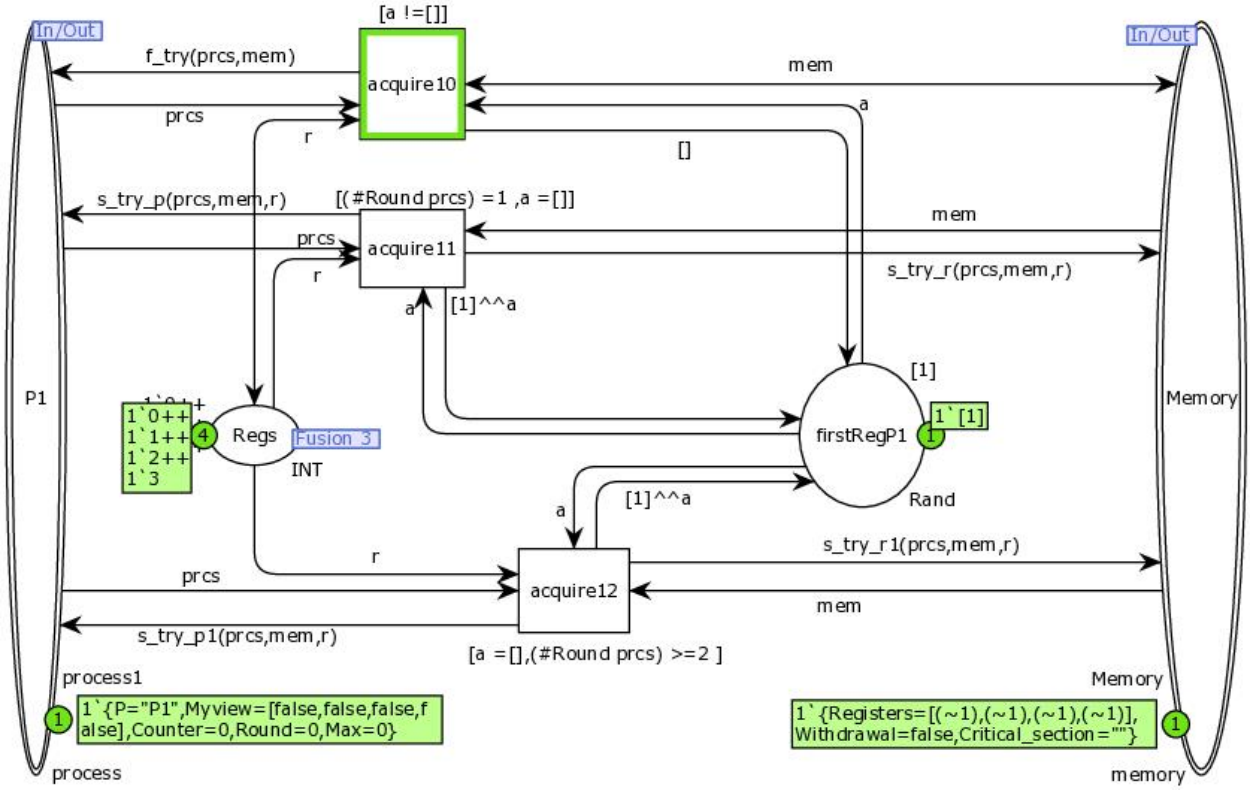

Fig. 3. The CPN model of the low level (the submodule Obtain_RegP1)

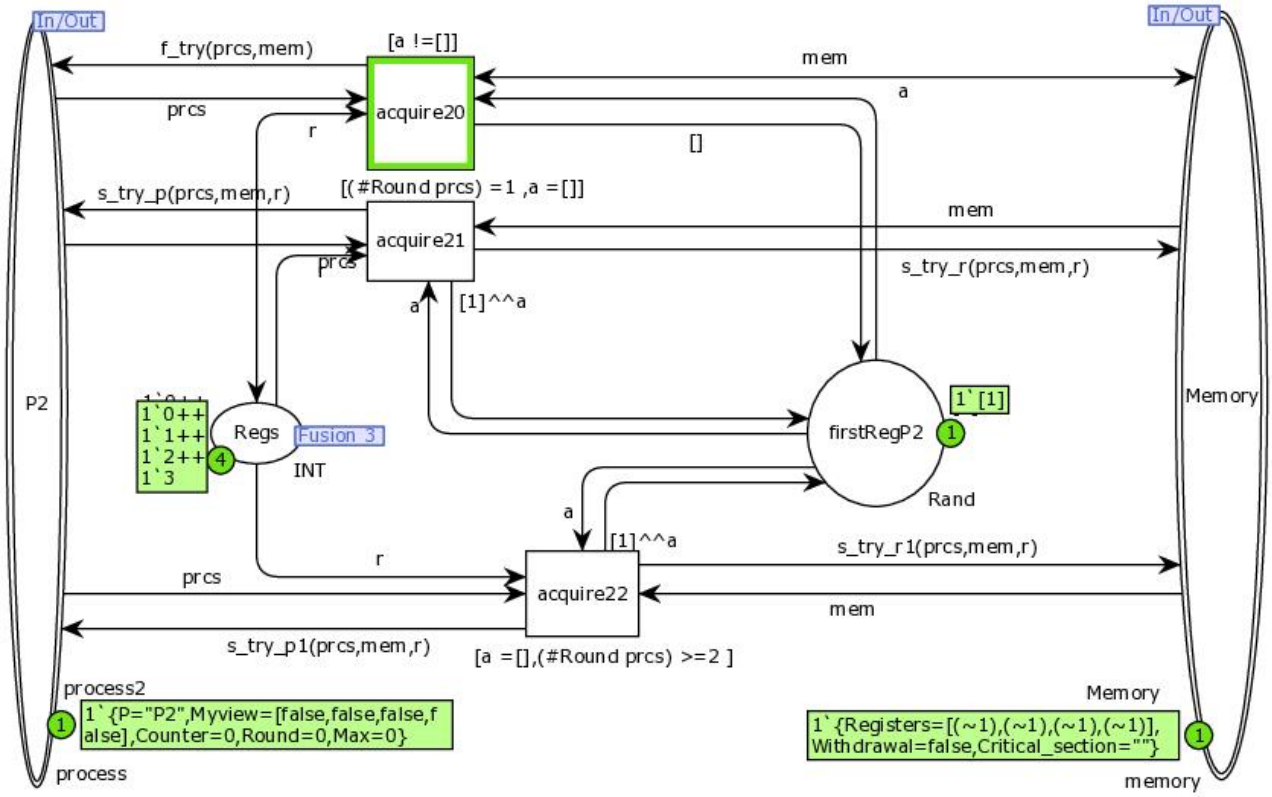

Fig. 4. The CPN model of the low level (the submodule Obtain_RegP2)

Journal of Distributed Computing and Systems(JDCS)
Vol.6, Issue 2, 2024, Pages 1-11

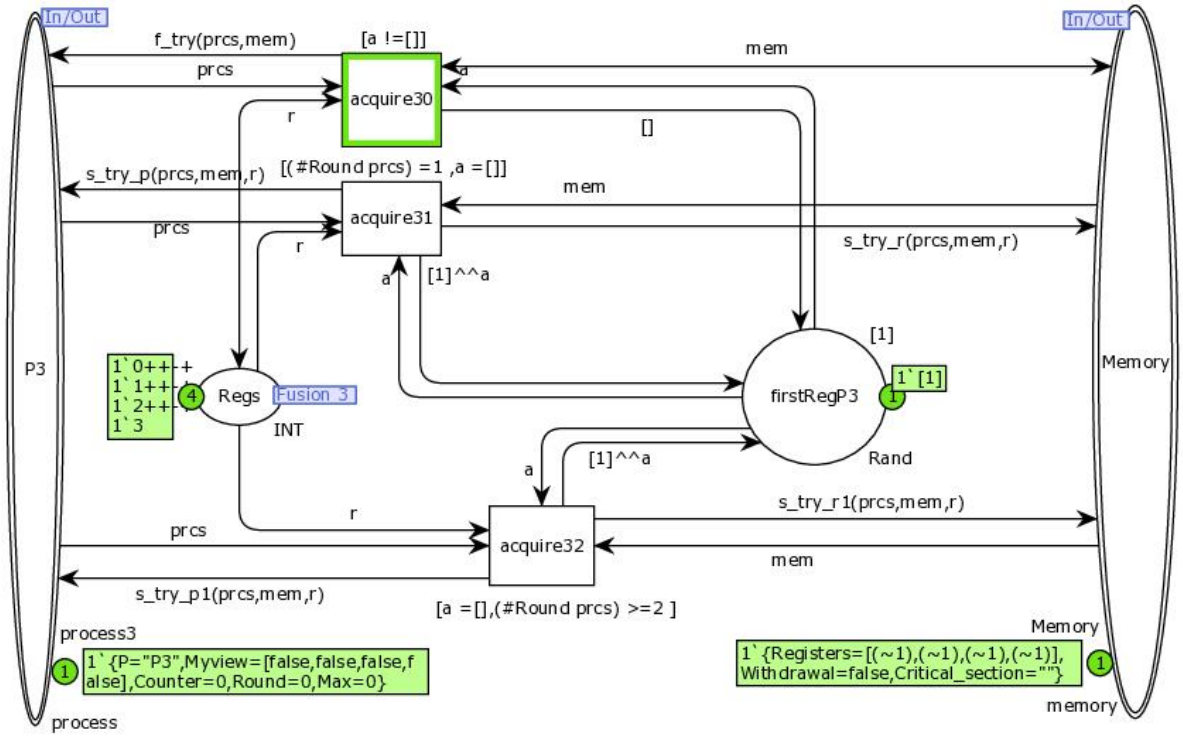

Fig. 5. The CPN model of the low level (the submodule Obtain_Reg

The place *Regs* is used as a fusion set in the submodule *Obtain_RegP1,* which is a copy of the registers, which changes at the same time at the low level if it is changed at the top level. The transitions called *acquire10*, *acquire11*, and *acquire12* represent the operation of process P_1 to own the registers in different rounds. When *Round* < *Max*, the transition *acquire10* is enabled, and when *Round* = *1*, the transition *acquire11* is enabled. When *Round* ≥ *2,* the transition *acquire12* is enabled. We define a place called *firstRegP1*, whose initial marking is a list containing the number one by default.

When place *Regs* contains all the registers, it means that no process has yet owned any of the registers; therefore, the place *firstRegP1* includes a token, and the transition *acquire10* is enabled and can fire. By firing the transition *acquire10*, the process P_1 owns the first register, and the *function f_try*() on the output arc of the transition *acquire10* updates the variables in place *P1* and increases the variable *Round*. The initial value of *Round* in the initial marking of place *P1* is equal to zero, and with its increase, the value of Round becomes equal to one. Also, *b*y firing the transition *acquire10*, the list related to the place *firstRegP1* is empty.

Therefore, in this step, the guard condition of the transition *acquire11* is met, and this transition is enabled and fires. By firing the transition *acquire11*, the process P_1 owns the second register, and the function *s_try_p*() on the output arc of the transition *acquire11* increases the variable *Round,* it becomes *Round* ≥ *2,* and the transition *acquire12* is enabled and fires. Thus, in each step, the value of the variable *Round* increases, and when transition *acquire12* is enabled, the process P_1 acquires a register from the registers that have not yet been acquired.

We design Fig. 4 like the submodule *Obtain_RegP1*, except that we define four places called *acquire20*, *acquire21*, *acquire22*, and *firstRegP2* instead of four places *acquire10*, *acquire11*, *acquire12*, and *firstRegP1*, respectively. All operations mentioned for the submodule *Obtain_RegP1* also apply to the submodule *Obtain_RegP2*. Also, we design Fig. 5 like the submodule *Obtain_RegP1*, except that we define four places called *acquire30*, *acquire31*, *acquire32*, and *firstRegP3* instead of four places *acquire10*, *acquire11*, *acquire12*, and *firstRegP1*, respectively. All operations mentioned for the submodule *Obtain_RegP1* also apply to the submodule *Obtain_RegP3*.

V. STATE SPACE ANALYSIS AND VERIFICATION OF PROPERTY OF MODEL

This section reports the results of the state space analysis and then proves the correctness of the ME algorithm, which means that it displays that only one process accesses the memory at any time. In addition, it shows that the processes do not coordinate to identify the registers related to the memory.

State space analysis results: We can use numerous analysis techniques such as sweep line analysis, language analysis, simulation, and state space analysis to analyze the model produced by the colored Petri nets [8]. We create the model's state space diagram with the CPN model checking tool and

then analyze the model using the state space analysis technique. The report of state space analysis of the model of the ME algorithm illustrates that the graph consists of 34155 nodes and 69977 arcs, the graph generation time is 216 seconds, and the graph status is full.

Verification of the mutual exclusion property: According to the mutual exclusion property of the ME algorithm, other processes are not allowed to enter their critical section when a process is in its critical section. Figures 6, 7, and 8 illustrate the verification steps for this property for P_1, P_2, and P_3, respectively. Figure 6 illustrates the CPN query codes for analysis of the mutual exclusion property for P_1. According to Code 1, in node 1 of the state space graph, the place *Regs* contains all the default registers. Code 2 indicates that in node 19097, the processes own all the desired registers, whereas node 19097 is reachable from node 1 (Code 3). Code 4 indicates that P_1 owns registers 4, 3, 2, and 1, respectively, which means that it has acquired memory and entered the critical section in node 19097. Codes 5, 6, and 7 indicate the status of processes P_1, P_2, and P_3, respectively, in node 19097; Where the value of the variable *Myview* related to P_1 is true, and the value of the variable *Myview* related to other processes is false. Code 8 indicates the number of nodes that P1 passes to enter the critical section. According to Fig. 7 and Fig. 8, the processes P_2 and P_3 have entered their critical section in nodes 22974 and 14227, respectively. By analyzing the CPN query codes in Figures 6, 7, and 8, we conclude that when a process enters its critical section, other processes are not allowed to enter it.

Verification of the anonymous memory: By investigating the variable Registers in Code 4 of Figures 6, 7 and 8, we observe that the knowledge of process P_1 about the ID of the registers is different from the knowledge of processes P_2 and P_3 about the ID of the same registers. In other words, process P_1 recognizes register 4 as the first register, process P_2 recognizes register 4 as the fourth register, and process P_3 recognizes register 4 as the second register. Thus, the condition of anonymous memory is satisfied.

Deadlock Detection: Figure 9 illustrates that this algorithm is prone to deadlock. Let us consider a scenario of model execution where two processes, P_2 and P_3, own two registers each. If both processes P_2 and P_3 intend to try to obtain other registers, according to the diagram presented in Fig. 1, the condition $Round_i \geq 2$ is satisfied for both processes, and both try to acquire other registers. They haven't released any of the registers they have already acquired. As a result, the algorithm faces a deadlock. According to Codes 6 and 7 of Fig. 9, the processes P_2 and P_3 each own two registers located in *Round = 2*. According to Code 2, all the desired registers are finished, but no process has entered the critical section yet (Code 4).

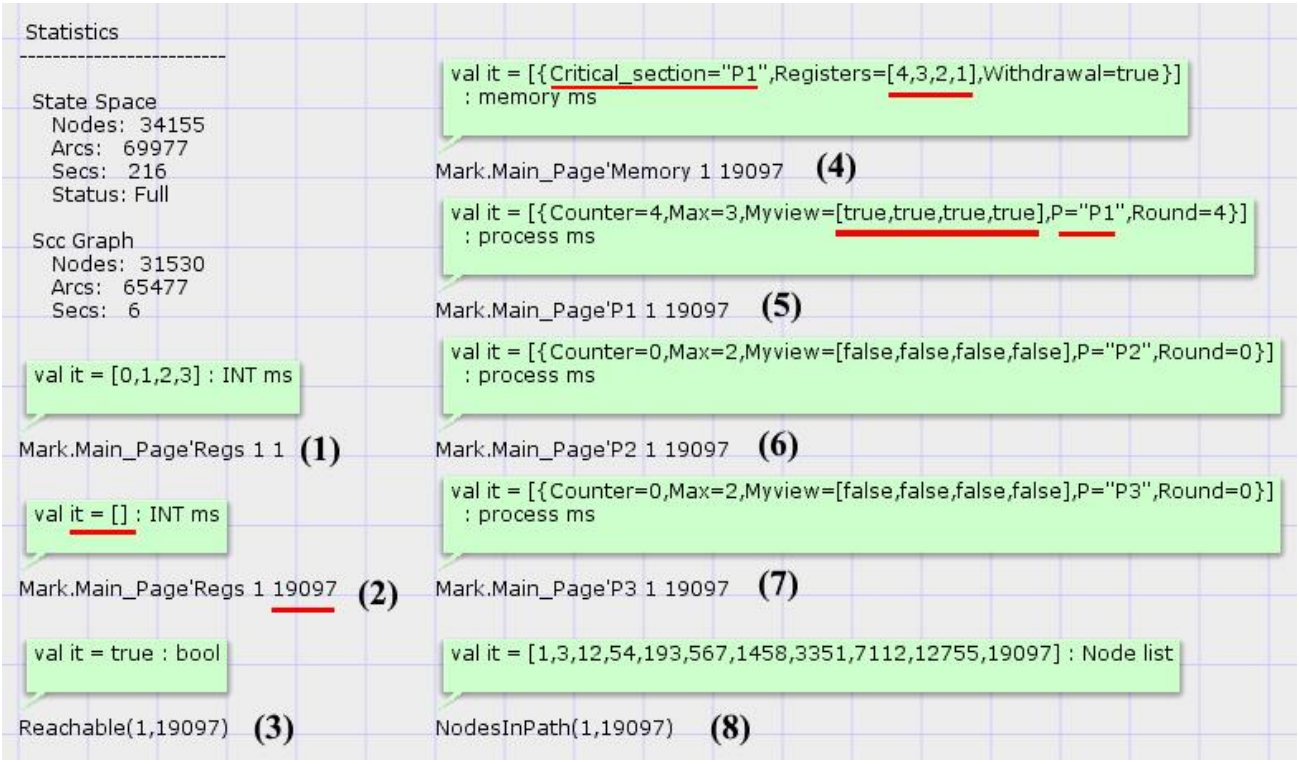

Fig. 6. The state space analysis: the CPN query codes for analysis of the mutual exclusion property for P_1

Journal of Distributed Computing and Systems(JDCS)
Vol.6, Issue 2, 2024, Pages 1-11

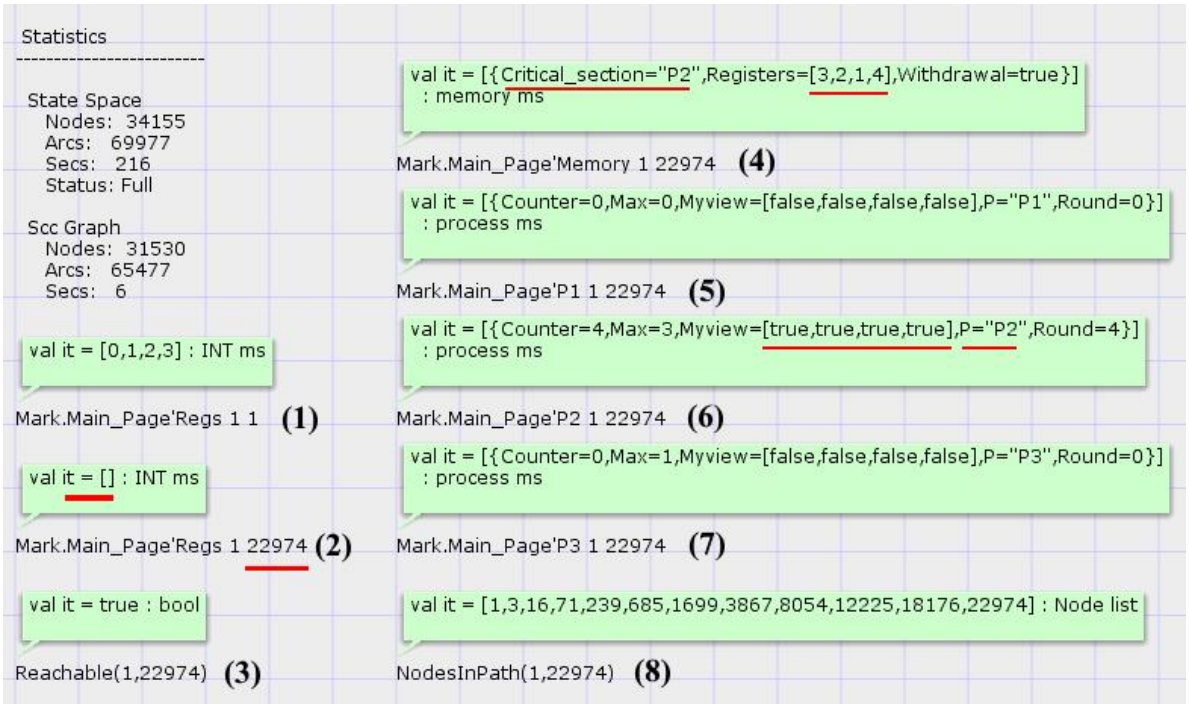

Fig. 7. The state space analysis: the CPN query codes for analysis of the mutual exclusion property for P_2

Fig. 8. The state space analysis: the CPN query codes for analysis of the mutual exclusion property for P_3

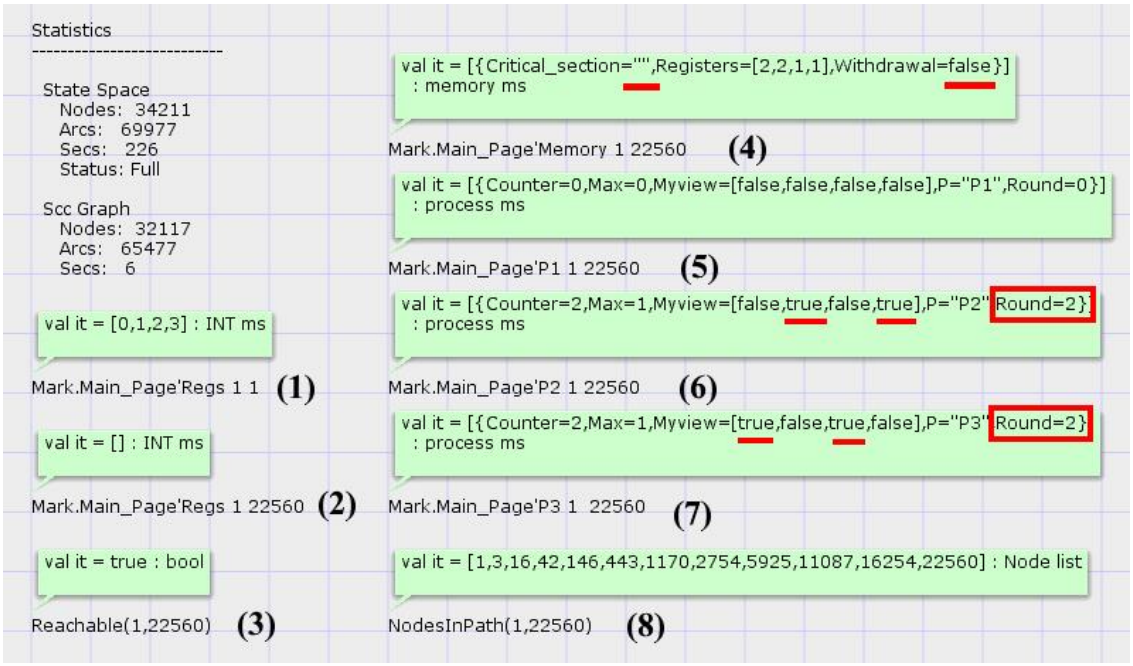

Fig. 9. The state space analysis: the CPN query codes for analysis of deadlock

VI. CONCLUSION AND FUTURE WORK

This study models and validates the mutual exclusion (ME) algorithm, which plays a vital role in distributed systems using formal methods. The algorithm ensures that desired behaviors and properties are upheld in distributed environments. The term "fully anonymous" means that processes and memory cannot be distinguished from each other. The research employs Colored Petri Nets (CPN) hierarchically to model the ME algorithm, incorporating low and high levels. Analysis of the state space diagram indicates that typically, only one process accesses the critical section while others wait.

Furthermore, the study demonstrates that different processes identify a particular memory register with distinct identifiers. However, it highlights a potential deadlock scenario where two processes simultaneously acquire an equal number of registers and cannot release them, resulting in deadlock. This paper signifies the first modeling and validation of a fully anonymous mutual exclusion algorithm using a mechanical proof method, showcasing the innovation of utilizing colored Petri nets for such purposes. The presented model encompasses the fundamental property of full anonymity and can serve as a basis for modeling similar distributed algorithms, thereby solidifying its significance as a reference framework in this field.

VII. MODEL AVAILABILITY

The CPN model associated with this article is available via request from the authors.

REFERENCES

[1] M. Raynal and G. Taubenfeld, "Mutual exclusion in fully anonymous shared memory systems," Inf. Process. Lett., vol. 158, p. 105938, 2020.

[2] M. Raynal and G. Taubenfeld, "A visit to mutual exclusion in seven dates," Theor. Comput. Sci., vol. 919, pp. 47-65, 2022.

[3] R. Mateescu and W. Serwe, "Model checking and performance evaluation with CADP illustrated on shared-memory mutual exclusion protocols," Sci. Comput. Program., vol. 78, no. 7, pp. 843-861, 2013.

[4] M. L. Neilsen, "Model Checking Token-Based Distributed Mutual Exclusion Algorithms," in Proc. PDPTA, 2009, pp. 10-16.

[5] F. Cicirelli, L. Nigro, and P. F. Sciammarella, "Model checking mutual exclusion algorithms using u ppaal," in Proc. Software Engineering Perspectives and Application in Intelligent Systems: Proceedings of the 5th Computer Science On-line Conference 2016 (CSOC2016), Vol 2 5, 2016: Springer, pp. 203-215.

[6] K. Zhao, V. Margapuri, and M. Neilsen, "Model Checking Mutual Inclusion and Mutual Exclusion Algorithms," in Proc. ISCA 30th International Confer, vol. 77, pp. 60-69, 2021.

[7] K. Jensen, L. M. Kristensen, Coloured Petri nets: modelling and validation of concurrent systems, Springer Science & Business Media, 2009.

[8] V. Sahota, M. Li, M. Hadjinicolaou, "Modeling Scalable Grid Information Services with Colored Petri Nets," International Journal of Grid and High Performance Computing (IJGHPC), vol. 2, no. 1, pp. 51-68,Jan. 2010.

[9] M. Raynal and G. Taubenfeld, "A visit to mutual exclusion in seven dates," Theor. Comput. Sci., vol. 919, pp. 47-65, 2022.

[10] S. Dhoked, W. Golab, and N. Mittal," Brief Announcement: On Solving Recoverable Mutual Exclusion Under System-Wide Failures," in Proc. the 35th ACM Symposium on Parallelism in Algorithms and Architectures, 2023, pp. 287-290.

[11] J. J. Daymude, A. W. Richa, and C. Scheideler, "Local mutual exclusion for dynamic, anonymous, bounded memory message passing systems," arXiv preprint arXiv:2111.09449, 2021.

[12] D. Imbs, M. Raynal, and G. Taubenfeld, "Election in fully anonymous shared memory systems: tight space bounds and algorithms," in

International Colloquium on Structural Information and Communication Complexity, 2022: Springer, pp. 174-190.

[13] A. D. Kshemkalyani and M. Singhal, Distributed computing: principles, algorithms, and systems. Cambridge University Press, 2011.

[14] S. Navlakha and Z. Bar-Joseph, "Distributed information processing in biological and computational systems," Commun. ACM, vol. 58, no. 1, pp. 94-102, 2014.

[15] M. L. Neilsen, "Model Checking Prioritized Token-Based Mutual Exclusion Algorithms," in Proc. the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA), 2013: The Steering Committee of The World Congress in Computer Science, Computer Engineering and Applied Computing (WorldComp), p. 74.

[16] A. Mostéfaoui, M. Petrolia, M. Raynal, and C. Jard, "Atomic read/write memory in signature-free byzantine asynchronous message-passing systems," Theory Comput. Syst., vol. 60, pp. 677-694, 2017.

[17] S. Baarir, J. Sopena, and F. Legond-Aubry, "Verification of a hierarchical generic mutual exclusion algorithm," in Formal Techniques for Networked and Distributed Systems–FORTE 2008: 28th IFIP WG 6.1 International Conference Tokyo, Japan, June 10-13, 2008 Proceedings 28, 2008: Springer, pp. 99-115.

[18] K. Ogata and K. Futatsugi, "Formally modeling and verifying Ricart&Agrawala distributed mutual exclusion algorithm," in Proc. Second Asia-Pacific Conference on Quality Software, 2001: IEEE, pp. 357-366.

[19] K. Zhao, V. Margapuri, and M. Neilsen, "Model Checking Mutual Inclusion and Mutual Exclusion Algorithms," in Proc. of ISCA 30th International Confer, vol. 77, pp. 60-69, 2021.

[20] M. Hendriks et al., "UPPAAL 4.0," in Proc. Third International Conference on the Quantitative Evaluation of Systems-(QEST'06), 2006: IEEE, pp. 125-126.

How to cite: L.NamvariTazehkand, S. Pashazadeh. **Modeling and Formal Verification of a Distributed Mutual Exclusion Algorithm**, Journal of Distributed Computing and Systems(JDCS), Vol 6, Issue 2, Page 1 - 11, 2024.

Leila NamvariTazehkand is a Ph.D. Candidate in computer engineering at the Faculty of Electrical and Computer Engineering, University of Tabriz, Tabriz, Iran. She received her Bachelor's and Master's degrees in computer engineering in 2014 and 2019 respectively from the Payam Noor University of Tabriz and the University of Tabriz. Her research interests include distributed algorithms and systems, modeling and formal verification, and reliability analysis.

Saeid Pashazadeh received the B.Sc. degree in computer engineering from the Sharif University of Technology, Tehran, Iran, in 1995; the M.Sc. and Ph.D. in computer engineering from the Iran University of Science and Technology, Tehran, Iran, in 1998, 2010; He is currently a professor in the Department of Information Technology, the Faculty of Electrical and Computer Engineering, University of Tabriz, Tabriz, Iran. His research interests include software engineering, modeling and performance evaluation, distributed systems.