Model Checking Token-Based Mutual Inclusion and Mutual Exclusion Algorithms

Kai Zhao, Venkat Margapuri, and Mitchell Neilsen

Department of Computer Science

Kansas State University

[kaizhao@ksu.edu](mailto:kaizhao@ksu.edu), [marven@ksu.edu](mailto:marven@ksu.edu), [neilsen@ksu.edu](mailto:neilsen@ksu.edu)

Abstract

The concepts of mutual inclusion and mutual exclusion are critical for concurrency control in distributed systems. Mutual exclusion is a property which ensures that at most one process can execute in its critical section at any given time. For example, other processes are not allowed to enter their critical sections when a given process is updating a shared variable in its critical section. If up to *k* processes can enter their critical sections, this is called *k-exclusion*. In contrast, mutual inclusion imposes restrictions on processes from leaving their critical sections. For example, to ensure reliability in a server farm, a certain number of servers may need to be available to service requests. If at least *m* processes must be available, this is called *m-inclusion*. Model checking is essential to verify and validate correctness and safety properties of distributed algorithms. The paper presents token-based models that can be used to verify and validate *k-mutual exclusion* and *l-mutual inclusion* algorithms where *k* refers to the maximum number of processes in their critical sections and *l* is the minimum number that must remain in their critical sections. Verification criteria includes the maximum number of messages that must be exchanged to enter or leave a critical section, deadlock freedom, and timing parameters. In addition, a model that includes both *k-exclusion* and *m-inclusion* is presented to demonstrate the feasibility of evaluating both mutual exclusion and mutual inclusion in the same model. Models are developed in UPPAAL, an environment for modeling, validation, and verification of real-time systems represented using timed automata. Finally, we propose some new self-stabilizing algorithms and models for 1-exclusion and 2-exclusion.

1. Introduction

In modern systems, many problems require multitasking – multiple tasks running at the same time. Multiple tasks (or processes) in a distributed system may access a resource simultaneously or execute a given function at the same time. The instructions used to update a shared resource are commonly referred to as a critical section (CS). If execution within each critical section is not controlled, more than one process may try to update a shared resource at the same time which may cause inconsistencies in the shared resource. Algorithms that guarantee at most one process is allowed in its CS at any given time are called mutual exclusion algorithms. Mutual exclusion is an example of concurrency control. Dijkstra first raised this question in 1965 [1]. In some cases, up to *k* processes can execute in their critical sections simultaneously, this is called *k-exclusion*. In the simplest case, 1*-exclusion* is just *mutual exclusion*. Distributed mutual inclusion is complementary to distributed mutual exclusion. Mutual exclusion restricts processes entering their critical sections, while mutual inclusion restricts processes from leaving their CS. For 1*-inclusion*, also called mutual inclusion, at least one task must be in its CS at all times. For instance, to provide higher system availability, we may require more than one server task to be available to process requests. Due to this complementary relationship, it is easier to speculate on algorithms to solve the inclusion problem based on similar principles after the problem of mutual exclusion is solved. Combining *k*-exclusion and *m*-inclusion, we can generalize the mutual exclusion and mutual inclusion, which is expressed as a (*k, m*) – exclusion, inclusion algorithm. For example, (1,0) – exclusion, inclusion is ordinary mutual exclusion. In a distributed environment, message passing is typically used to achieve mutual exclusion [2]. Distributed mutual exclusion algorithms can be divided into centralized algorithms and distributed algorithms. Distributed algorithms can be divided into token-based algorithms and permission-based algorithms [3]. This paper focuses on token-based distributed algorithms. Possession of a token allows a process to enter or leave its critical section. Different tokens are used for inclusion and exclusion.

UPPAAL is a toolbox for verification of real-time systems jointly developed by Uppsala University and Aalborg University. It has been applied successfully in case studies ranging from communication protocols to multimedia applications [5]. The tool is designed to verify systems that can be modeled as timed automata networks extended with integer variables, structured data types, and channel synchronization. The timed automata are finite state machines with real-valued clocks. It uses a dense time model in which the time variable is evaluated as a real number, and all clocks are synchronized. In UPPAAL, a system is simulated as a parallel network of several such timed automata. UPPAAL includes a simulator for random and guided simulation of the timed automata. A model checker is provided to verify safety and liveness properties. Properties are expressed in a subset of Computation Tree Logic (CTL). For an invariant property p, the expression A[] p means that on all paths (A), the property p is always ([]) satisfied. For a liveness property p, or to test for reachability, the expression E<> p means that on some path (E) the property p is eventually (<>) satisfied.

Several mutual exclusion algorithms and mutual inclusion algorithms form the basis for this work. The algorithms in the area of mutual exclusion are divided into two categories, token-based and permission-based. The works [1, 3, 6, 9] are permission-based where in a process is restricted from entering its critical section until permission is obtained from a quorum of processes. For token-based algorithms, such as [9, 11], a process is restricted from entering the critical section until a token within the scope of the system is obtained, giving the process the right to enter its critical section. For k-exclusion, a set of k tokens can be passed in a logical ring to enforce k-exclusion [9]. Thiare extended the algorithm to achieve self-stabilization [8]. The focus of this paper will be on token-based ring algorithms for k-exclusion and m-inclusion.

1. Token-based Ring Algorithms

Token-based ring algorithms are used on networks logically arranged as rings. The token in the network is the entity that controls concurrency in the network to ensure that the system is devoid of any race conditions, a condition where multiple tasks (or processes) try to access and modify the same shared resource. UPPAAL models are designed for both k-exclusion and m-inclusion.

**Assumptions**

Both of the algorithms rely on a common set of assumptions:

1. No loss of messages between different processes (also called nodes or tasks).
2. No delay in transmission of tokens between different processes (nodes).
3. Messages arrive in order.
   1. Token Ring k-Exclusion Algorithm

The algorithm and correctness properties of the Token Ring *k*-Exclusion Algorithm [4, 6, 9, 10] are listed below:

**Algorithm**

1. All processes (nodes) form a logical ring structure. Tokens are passed between nodes in a clockwise (or counterclockwise) direction to a neighbor.
2. A node that receives a token has the right to enter its CS.
3. A token is transmitted to the next node after a node leaves its CS.
4. If a node does not need to enter its CS, any received token is passed directly to the next node without delay.
5. For *k*-exclusion, *k* **exclusion tokens** circulate in the ring.

**Properties**

1. Safety: At any instant, at most *k* nodes can be in their critical sections.
2. Liveness: A node that wants to enter its critical section should eventually be allowed to enter.
3. Fairness: Each node gets a fair chance to execute in its CS. Fairness is only biased by the logical ordering imposed by the ring. The property means the CS execution requests are executed in the order of their arrival (a logical clock determines time) in the system.
   1. Token Ring m-Inclusion Algorithm

The algorithm and correctness properties of the Token Ring m-inclusion algorithm [4] are as follows:

**Algorithm**

1. All processes (nodes) form a logical ring structure. The tokens pass between the nodes in a clockwise (or counterclockwise) direction to a neighbor.
2. A node that receives a token has the right to ***leave*** its CS.
3. A token is transmitted to the next node after a node ***enters*** its CS.
4. If a node does not need to leave its CS, any received token is passed directly to the next node without delay.
5. For *m*-inclusion, *m* **inclusion tokens** circulate in the ring.

**Properties**

1. Safety: At any instant, at least *m* tasks must be executing in their critical sections.
2. Liveness: A task that wants to leave its CS should eventually be allowed to leave.
3. Fairness: Each process gets a fair chance to execute outside the CS. Fairness is only biased by the logical ordering imposed by the ring. This property means that requests to leave CS are executed in the order of their arrival (a logical clock determines time) in the system.
4. Model Checking and Analysis

3.1 k-Exclusion Model for Model Checking

The distributed model is a ring network modeled in UPPAAL with five nodes where at most two nodes (k = 2) may simultaneously enter the critical section. The variable ‘nb’ records neighbors of each node, and three channels to communicate with a passive observer used to verify properties. The variable ‘count’ records the number of nodes in the CS. The structure ‘S’ records the information at each node, including ‘requesting’, ‘numTokens’ and ‘inCS’. Initially, nodes 0 and 2 hold tokens. Each of the nodes starts in the Non-CS state. The model description in UPPAAL is as shown in Figure 1.

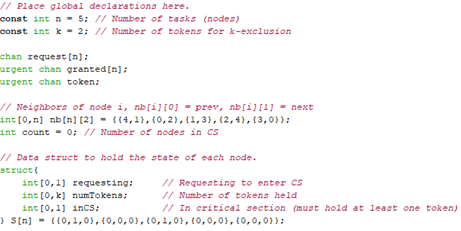


Figure 1: Model Description for k-Exclusion

The model is defined using two templates: Node and Observer. The Node template provides the state machine that models the k-exclusion algorithm, as shown in Figure 2. In the Non-CS state, if a token reaches a node that does not request for one, the token is passed to the node’s neighbor. However, if a token reaches a requesting node, the node may enter the CS. Whenever a node enters its CS, it sets variables S[id].inCS to 1 and S[id].requesting to 0. In the event a node in its CS receives another token, the token is passed to the neighboring node. The model implemented as part of the work restricts the amount of time that a node may spend in the CS to 10-time units. At the end of the 10-time units, the node leaves the CS, passes the token to the neighboring node and sets variable, S[id].inCS to 0. The count of the number of nodes in the CS is incremented when a node enters its CS, and it is decremented when a node leaves its CS.

The Observer template models a process used to verify a liveness property of the model. It is used to verify that a node requesting access to a token receives the token in at most 40-time units. As shown in Figure 3, the observer can enter the ‘BAD’ state if a requesting node does not receive the token within 40-time units. Each node has it’s own observer, and each observer uses its own local clock x to time the delay between a node requesting and entering its CS.

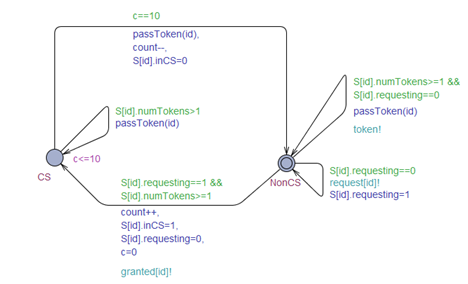


Figure 2: Model of Node for k-Exclusion

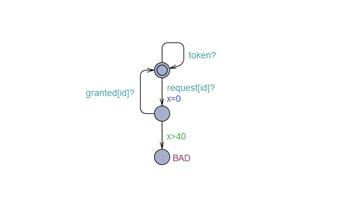


Figure 3: Model of Observer for k-Exclusion

3.1.1 Verification of Properties

Table 1: Properties Verified for k-Exclusion Model

|  |  |  |
| --- | --- | --- |
| Property | Result | Justification |
| A[] (not deadlock) | satisfied | No deadlock in the system |
| E<> (N2.CS && token[2] == 1) | satisfied | On some path, eventually node 2 may enter the CS |
| A[] (count <= k) | satisfied | At most k nodes may enter their CS at same time |
| E<> (N2.CS && N1.CS) | satisfied | Two nodes can enter their CS, k = 2 |
| A[] (not Observer.BAD) | satisfied | A node requesting to enter waits for at most 40-time units before entering. |

* 1. (k, m)-Exclusion, Inclusion Model Checking

In k-exclusion, the token meant for exclusion is used to enter the CS. The idea of m-inclusion is similar to that of k-exclusion except that an inclusion token is acquired to leave the CS. Figure 4 shows the declarations for (3, 2)-Exclusion, Inclusion where at least two nodes and at most three nodes may be in their CS. The variable ***n*** indicates the number of nodes, ***k***, the number of tokens for *k*-exclusion and ***m***, the number of tokens for *m*-inclusion equal to (***n – m***). Note that since we want two nodes to always be in their critical sections, we limit the number of inclusion tokens to *n-m*.

In the model as shown in Figure 6, two types of tokens are sent with calls to passToken() and passOutToken(). PassToken() passes tokens to allow a node enter the CS and passOutToken() passes tokens to allow a node to leave the CS. The idea is that a node that wishes to enter the CS sets the variable, ‘requesting’, to 1. Upon receiving the ‘enter’ token as it passes through the network, the node enters the CS. Similarly, a node that wishes to exit the CS sets the variable, ‘requestingOut’, to 1 and exits the CS upon receiving the ‘exit’ token as it passes through the network. The ‘requesting’ and ‘requestingOut’ variables are set to 0 upon an entry and exit of the CS respectively.

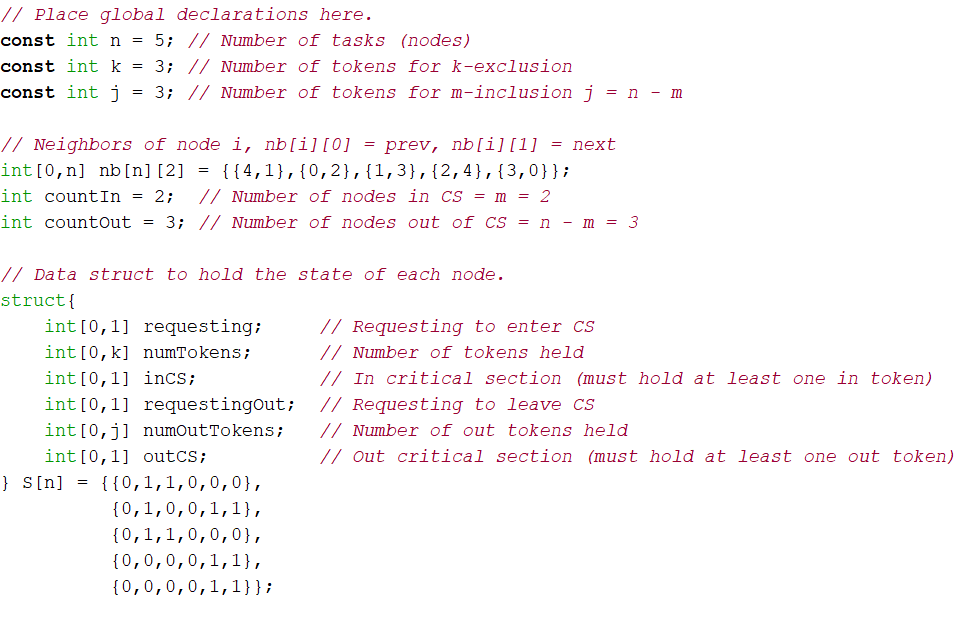


Figure 4: Model Description for (3, 2)-Exclusion, Inclusion with 5 nodes

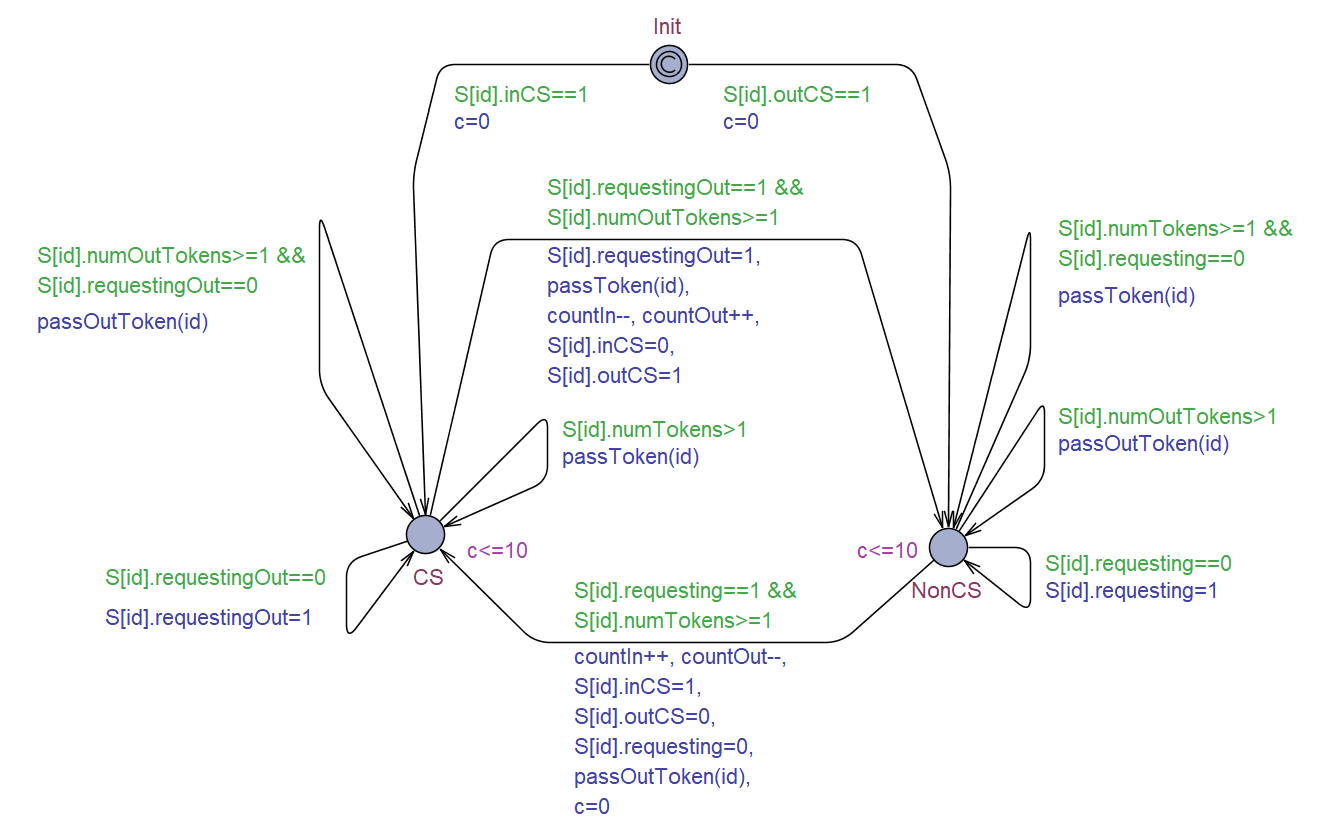
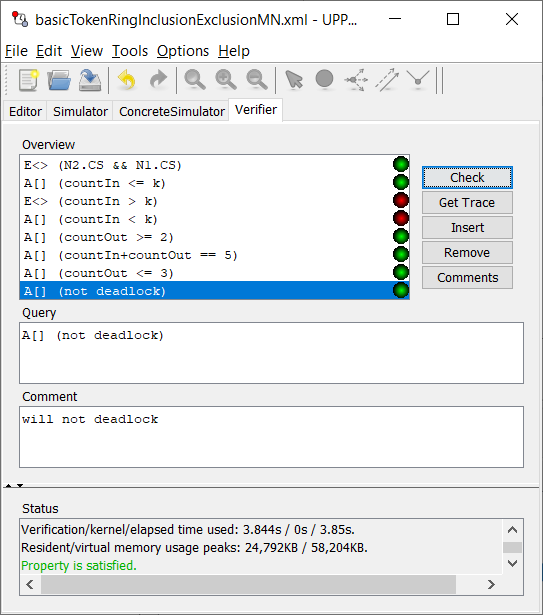


Figure 5: UPPAAL Model for (k, m)-Exclusion, Inclusion

* + 1. Verification of Properties



|  |  |  |
| --- | --- | --- |
| Property | Result | Justification |
| A[] (not deadlock) | satisfied | verifies that the system is devoid of deadlocks, a ‘safety’ property |
| E<> (N2.CS && token[2] == 1 | satisfied | A node that requests a token eventually receives it and enters the CS |
| A[] (count <= k && count >= (n-m) | satisfied | At least (n-m) nodes and at most k nodes may be in CS at a given point in time |
| E<> (N3.CS && N2.CS && N1.CS) | satisfied | k nodes enter the CS at the same time |
|  |  |  |

1. Self-Stabilizing Model

In the models described previously, the number of tokens at the time of initialization is the same as the target. For instance, in the 2-exclusion model, there are precisely two tokens initialized and both tokens are valid. However, it is possible that more than two tokens are initialized in which case there is one or more invalid tokens besides the two valid tokens. Alternatively, only one token may be initialized in which case the model lacks one valid token. The self-stabilizing approach [7] is used to solve the aforementioned problems. A node S is self-stabilizing concerning predicate P if it meets closure and convergence properties. The Dijkstra’s Self-Stabilizing Mutual Exclusion Algorithm [8] is used as the basis for the work and is briefly described as follows:

1. Assume that node 0 is the base node. S is the state of the node, L is the state of the node’s left neighbor, K is the number of states.
2. For the base-node, .
3. For other nodes, .

5.1 l-Exclusion Self-Stabilizing Model

In l-exclusion model, there are two cases of token initialization described as follows:

1. Precisely One Token: The case follows the Dijkstra’s algorithm described above and is relatively straight-forward to implement.
2. Multiple Tokens: It is required to ensure that only one token remains in the model eventually.

In order to account for the second possibility, the state of the node S which records the information of the nodes is redefined by adding three elements, described as follows:

1. Valid: A Boolean variable which records whether the token is valid and sets the first token seen by node 0 as valid.

2. Ready: A Boolean variable which records whether the whole ring is ready to allow some node enter its CS.

3. Seen: An integer variable bounded between 0 and 1 that records whether the base node has seen a valid token previously. 0 indicates not seen and 1 indicates seen.

In the model shown in Figure 7, regardless of the number of tokens initialized and the node at which the tokens are initialized, the first node seen by the base node is set to be valid and sent to its neighbor. The tokens seen later are deleted from the model as and when they arrive at the base node resulting in one and only one token remaining in the model eventually.

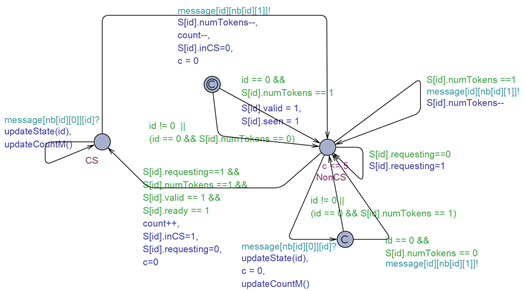


Figure 7: l-Exclusion Self-Stabilizing Model

* + 1. Verification of Properties

Table 3: Properties Verified for l-Exclusion Self-Stabilizing Model

|  |  |  |
| --- | --- | --- |
| Property | Result | Justification |
| A[] (not deadlock) | satisfied | Verifies that the system is devoid of deadlocks, a ‘safety’ property |
| A[] (count <= k) | satisfied | At most k nodes may be present in the CS at any given instant of time, a ‘safety’ property |
| A[] (countM > 15 => S[0].ready) | satisfied | Shows at least m + n + p messages are required for self-stabilization in the worst-case where m is the sum of the distance from each node with an initialized token, n is the node count and p is the number of invalid tokens |
| A[] (S[0].ready => countM > 6) | satisfied | The best-case scenario where at least m + n messages are required for self-stabilization where m is the distance between the base-node and the nearest node that holds a token |

5.2 2-Exclusion Self-Stabilizing Model using Byte Token

To realize self-stabilization and k-exclusion together, the use of logical rings is commonly employed. However, the approach requires that there be as many logical rings as k i.e. 2-exclusion requires 2 logical rings and so on. An efficient and optimal approach is the use of byte tokens i.e. the bitwise representation of a token for each of the nodes in the model. The bitwise representations are cascaded k times to achieve k-exclusion. For the work, 2-exclusion self-stabilization is demonstrated on a model with five nodes where the number of tokens initialized is five. The bitwise representations of the tokens 0, 1, 2, 3 and 4 are 000, 001, 010, 011 and 100 respectively. Since k = 2, the bitwise representations are cascaded twice yielding 000000, 001001, 010010, 011011 and 100100. The byte token expression essentially embeds two individual tokens within it where in the first three bits signify a token different from the last three bits. Depending upon the token a node chooses to manipulate, the node alters the corresponding part of the byte token expression. Figure 8 shows the self-stabilization model developed for 2-exclusion as part of the work.

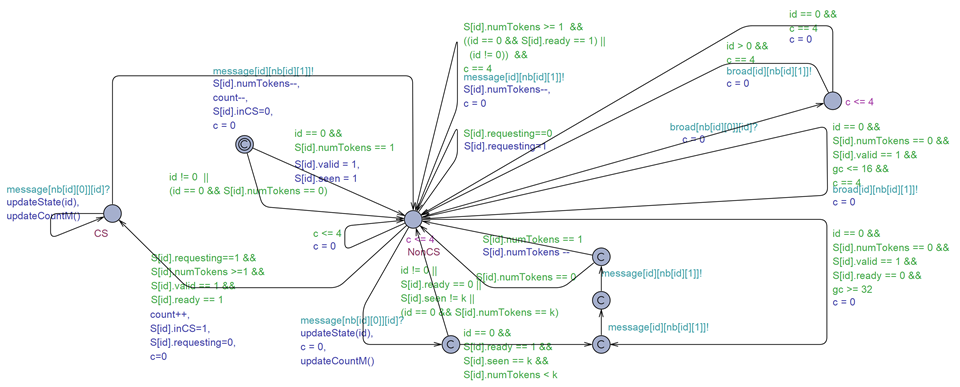


Figure 8: 2-Exclusion Self-Stabilizing Model using Byte Token

5.2.1 Verification of Properties

Table 4: Properties Verified for 2-Exclusion Self-Stabilizing Model using Byte Token

|  |  |  |
| --- | --- | --- |
| Property | Result | Justification |
| A[] (not deadlock) | satisfied | verifies that there is no deadlock in the system at all times, a ‘safety’ property. |
| E<> (N2.CS && token[2] == 1 | satisfied | each node eventually enters the CS upon requesting for a token. |
| A[] (count <= k) | satisfied | at most k nodes may enter the CS at the same time |
| A[] (countM > 7 => S[0].ready) | satisfied | The worst-case scenario in which m + p messages are required for self-stabilization where m is the sum of the distance from base-node to the two farthest nodes with tokens initialized and p is the count of invalid tokens |
| A[] (S[0].ready => countM > 4) | satisfied | Shows that m messages are required where m is the distance from base-node to the two nearest nodes that hold a token |

6 Conclusion

While the traditional centralized approach for mutual exclusion has the advantages such as high fairness, the approach suffers from problems such as single point of failure. As a result, the networks encounter a bottleneck in processing requests. In order to avoid the issues, distributed algorithms that combine mutual exclusion and inclusion are proposed to expand upon the scope of token-based algorithms. Besides, a self-stabilizing approach for k-exclusion using logical rings and byte tokens is proposed. The algorithms using the aforementioned approaches are implemented and verified using the software prototyping and verification tool UPPAAL to demonstrate the fault tolerance and stability of the algorithms. The implemented models are available at <https://github.com/VenkatMargapuri/Model-Checking-Token-Based-Algorithms> and any interest to further improve upon the work is welcome.

References

|  |  |
| --- | --- |
| [1] | Dijkstra, E. W. (1983). Solution of a problem in concurrent programming control. *Communications of the ACM*, *26*(1), 21-22. |
| [2] | Kshemkalyani, A. D., & Singhal, M. (2011). *Distributed computing: principles, algorithms, and systems*. Cambridge University Press. |
| [3] | Raynal, M. (1991). A simple taxonomy for distributed mutual exclusion algorithms. *ACM SIGOPS Operating Systems Review*, *25*(2), 47-50. |
| [4] | Kakugawa, H. (2015). Mutual inclusion in asynchronous message-passing distributed systems. *Journal of Parallel and Distributed Computing*, *77*, 95-104. |
| [5] | Cicirelli, F., Nigro, L., & Pupo, F. (2011). Modelling and Verification of Concurrent Programs Using UPPAAL. *ECMS*, *2011*, 525-533. |
| [6] | Neilsen, M. L. (2014). Real-Time Token-Based Mutual Exclusion Algorithms. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)* (p. 1). The Steering Committee of The World Congress in Computer Science, Computer Engineering and Applied Computing (WorldComp). |
| [7] | Afek, Y., Kutten, S., & Yung, M. (1997). The local detection paradigm and its applications to self-stabilization. *Theoretical Computer Science*, *186*(1-2), 199-229. |
| [8] | Dijkstra, E. W. (1982). Self-stabilization in spite of distributed control. In *Selected writings on computing: a personal perspective* (pp. 41-46). Springer, New York, NY. |
| [9] | Thiare, O. (2009). Several-Tokens Distributed Mutual Exclusion Algorithm in a Logical ring networks. In *International Conference on Machine Learning and Computing* (pp. 567-572). |
| [10] | Wu, W., Cao, J., & Raynal, M. (2007, December). A dual-token-based fault tolerant mutual exclusion algorithm for manets. In *International Conference on Mobile Ad-Hoc and Sensor Networks* (pp. 572-583). Springer, Berlin, Heidelberg. |
| [11] | Raymond, K. (1989). A tree-based algorithm for distributed mutual exclusion. *ACM Transactions on Computer Systems (TOCS)*, *7*(1) |

**Kakugawa ---** Self-stabilizing distributed algorithm for local mutual inclusion