EPiC Series in Computing
Volume 77, 2021, Pages 60–69
Proceedings of ISCA 30th International Conference on Software Engineering and Data Engineering

Model Checking Mutual Inclusion and Mutual Exclusion Algorithms

Kai Zhao, Venkat Margapuri, and Mitchell Neilsen
Department of Computer Science
Kansas State University
kaizhao@ksu.edu, marven@ksu.edu, 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 *m*-mutual inclusion algorithms where *k* refers to the maximum number of processes in their critical sections and *m* 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.

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*, mutual exclusion and mutual inclusion may be generalized and 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 [8]. Thiare extended the algorithm to achieve self-stabilization [9]. The focus of this paper will be on token-based ring algorithms for *k-exclusion* and *m-inclusion*.

2 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.

2.1 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.

2.2 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.

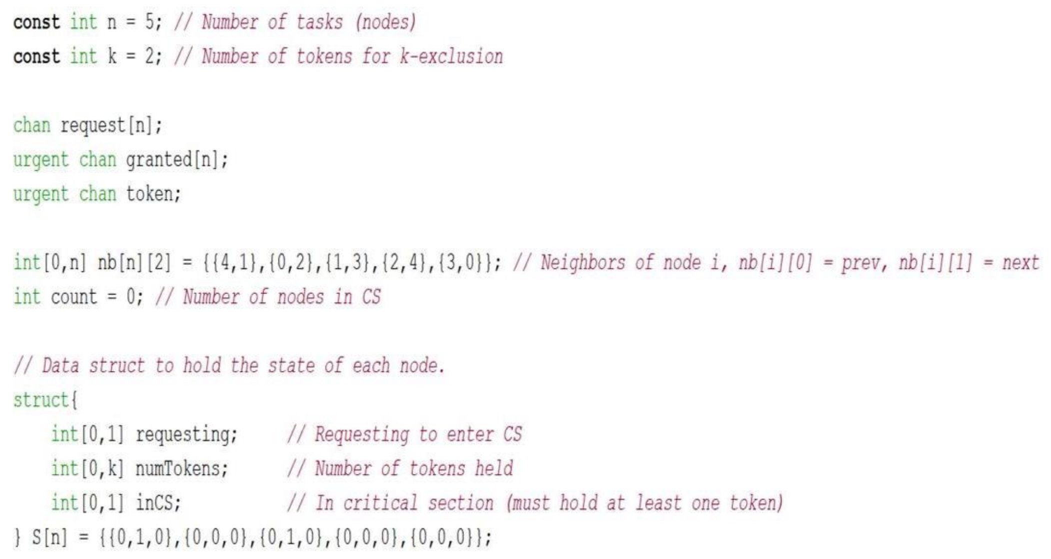
Properties

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.

3 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.



```
const int n = 5; // Number of tasks (nodes)
const int k = 2; // Number of tokens for k-exclusion

chan request[n];
urgent chan granted[n];
urgent chan token;

int[0,n] nb[n][2] = {{4,1},{0,2},{1,3},{2,4},{3,0}}; // Neighbors of node i, nb[i][0] = prev, nb[i][1] = next
int count = 0; // Number of nodes in CS

// Data struct to hold the state of each node.
struct{
    int[0,1] requesting;     // Requesting to enter CS
    int[0,k] numTokens;      // Number of tokens held
    int[0,1] inCS;           // In critical section (must hold at least one token)
} S[n] = {{0,1,0},{0,0,0},{0,1,0},{0,0,0},{0,0,0}};
```

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.

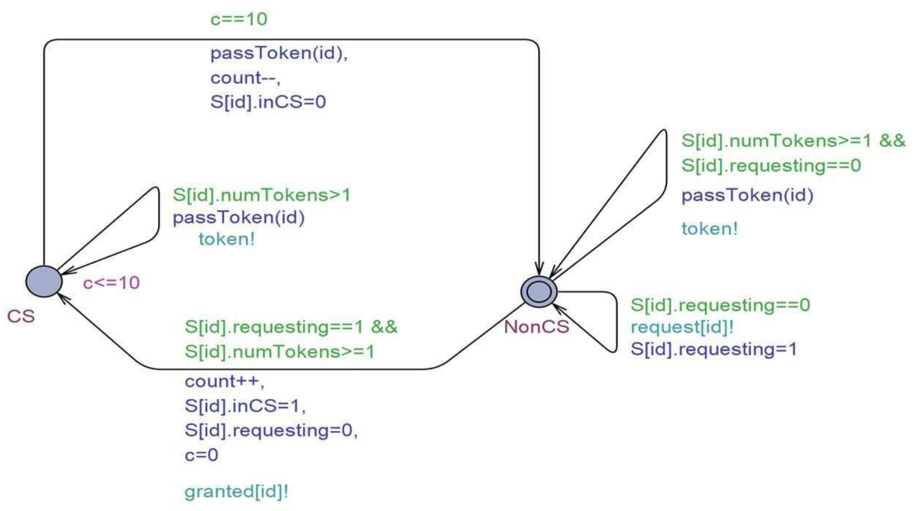


Figure 2: Model of Node for *k-Exclusion*

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 may enter the 'BAD' state if a requesting node does not receive the token within 40-time units. Each node has its own observer, and each observer uses its own local clock x to time the delay between a node requesting and entering its CS.

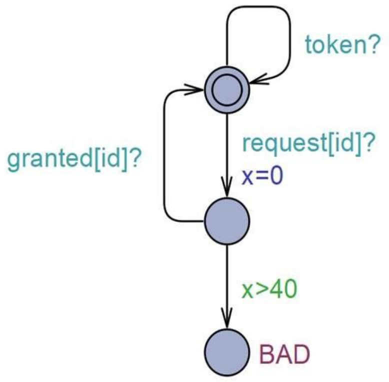
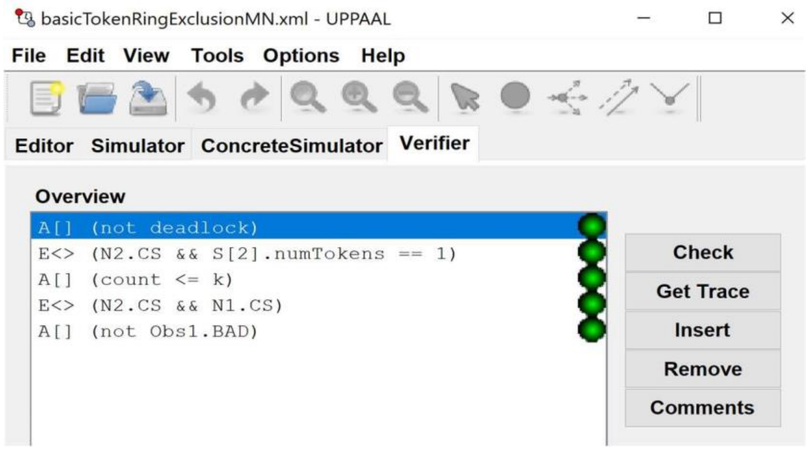


Figure 3: Model of Observer for *k-Exclusion*

3.1.1 Verification of Properties

The ETL for *k-Exclusion* is shown in Figure 4 and the justification for the verification of each of the properties is as described below.

1. A[] (not deadlock): No deadlocks in the system.
2. E<> (N2.CS && S[2].numTokens== 1): On some path, eventually node 2 may enter the CS
3. A[] (count <= *k*): At most *k* nodes may enter their CS at same time.
4. E<> (N2.CS && N1.CS): Two nodes can enter their CS, *k* = 2.
5. A[] (not Obs1.BAD): A node requesting to enter waits for at most 40-time units before entering the CS.

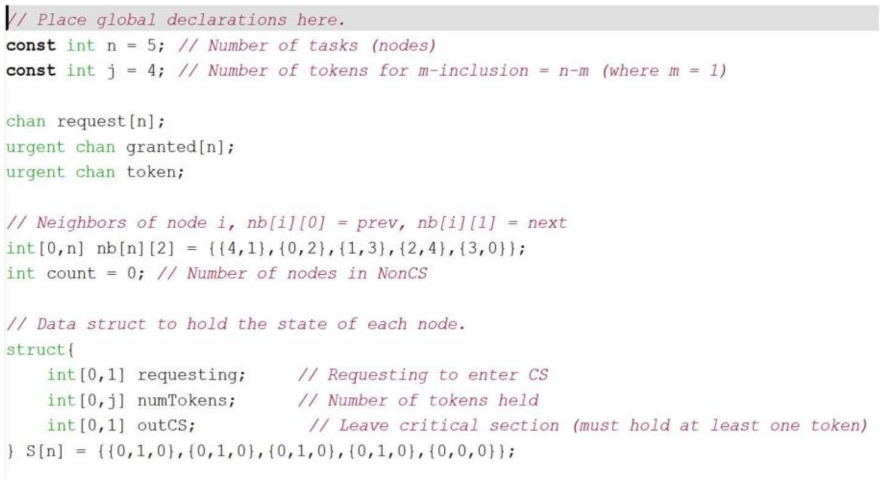

basicTokenRingExclusionMN.xml - UPPAAL
File Edit View Tools Options Help
Editor Simulator ConcreteSimulator Verifier
Overview
Check
Get Trace
Insert
Remove
Comments

```
A[] (not deadlock)
E<> (N2.CS && S[2].numTokens == 1)
A[] (count <= k)
E<> (N2.CS && N1.CS)
A[] (not Obs1.BAD)
```

Figure 4: *k-Exclusion* Properties Verified in UPPAAL

3.2 *m-Inclusion* Model Checking

The idea of *m-inclusion* is that at least *m* tasks must be present in the critical section at any given point in time. It is fair to state that inclusion is complementary to exclusion in terms of behavior. As a result, the distributed UPPAAL model for inclusion closely follows that of the model for exclusion and is a ring network with five nodes where at least one node (m = 1) is required to execute in the critical section at any given point in time. In other words, at most four nodes may leave the critical section simultaneously. Figure 5 shows the declarations for the UPPAAL model that achieves m-inclusion. The variable ‘count’ records the number of nodes out of the CS and variable ‘outCS’ tracks the status of the node. Initially, nodes 0, 1, 2 and 3 are initialized with tokens and each of the nodes starts in the CS state. Figure 6 shows the state machine that achieves m-inclusion. The state machine is similar to that of exclusion’s with the only difference being that the states ‘CS’ and ‘Non-CS’ are swapped. The observer to ensure that the model does not enter a bad state is the same as the observer used for *k-exclusion* as shown in Figure 3. This demonstrates that inclusion and exclusion are complementary to each other and lays the basis to achieving inclusion and exclusion in one model as described in section 3.3.



```
// Place global declarations here.
const int n = 5; // Number of tasks (nodes)
const int j = 4; // Number of tokens for m-inclusion = n-m (where m = 1)

chan request[n];
urgent chan granted[n];
urgent chan token;

// Neighbors of node i, nb[i][0] = prev, nb[i][1] = next
int[0,n] nb[n][2] = {{4,1},{0,2},{1,3},{2,4},{3,0}};
int count = 0; // Number of nodes in NonCS

// Data struct to hold the state of each node.
struct{
    int[0,1] requesting;     // Requesting to enter CS
    int[0,j] numTokens;      // Number of tokens held
    int[0,1] outCS;           // Leave critical section (must hold at least one token)
} S[n] = {{0,1,0},{0,1,0},{0,1,0},{0,1,0},{0,0,0}};
```

Figure 5: Declarations for *m-Inclusion* Distributed UPPAAL Model

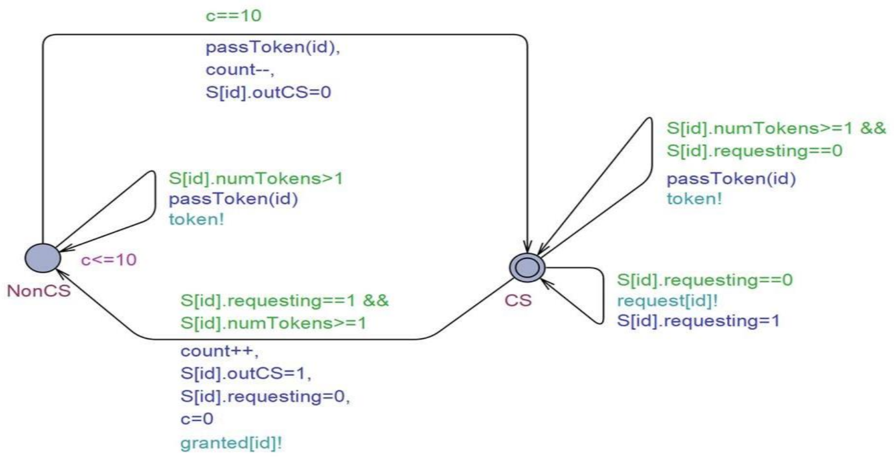
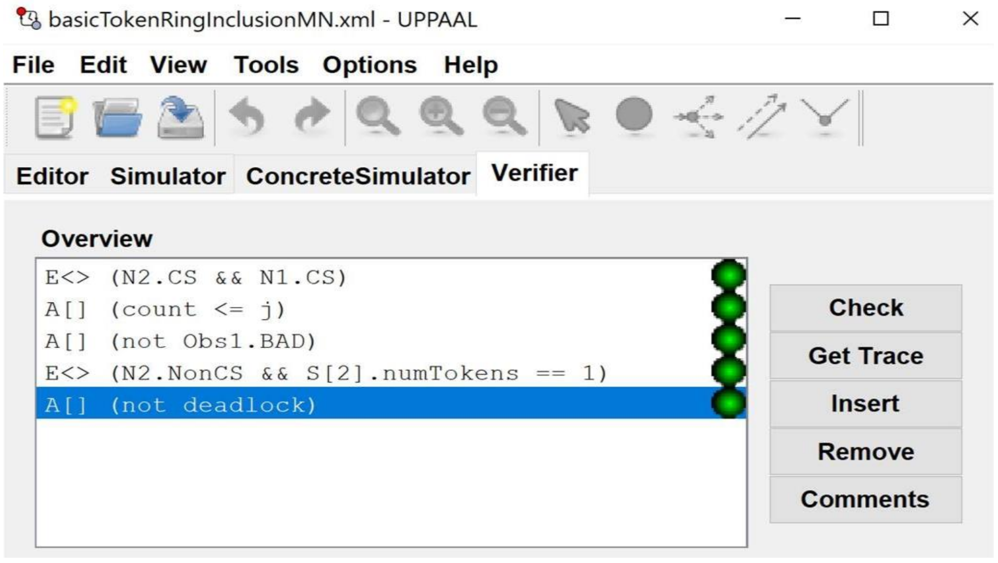


Figure 6: Model of Node for *m-Inclusion*

3.2.1 Verification of Properties

The ETL for *m-Inclusion* is shown in Figure 7 and the justification for the verification of each of the properties is as described below.

1. E<> (N2.CS && N1.CS): Eventually on some path, two (or j) nodes are allowed to enter their CS at the same time.
2. A[] (count <= j): On any given path, at most j nodes may enter the CS at the same time.
3. A[] (not obs1.BAD): On any given path, a node requesting to enter CS waits for at most 40time units before entering the CS.
4. A[] (not deadlock): On any given path, there are no deadlocks in the system.



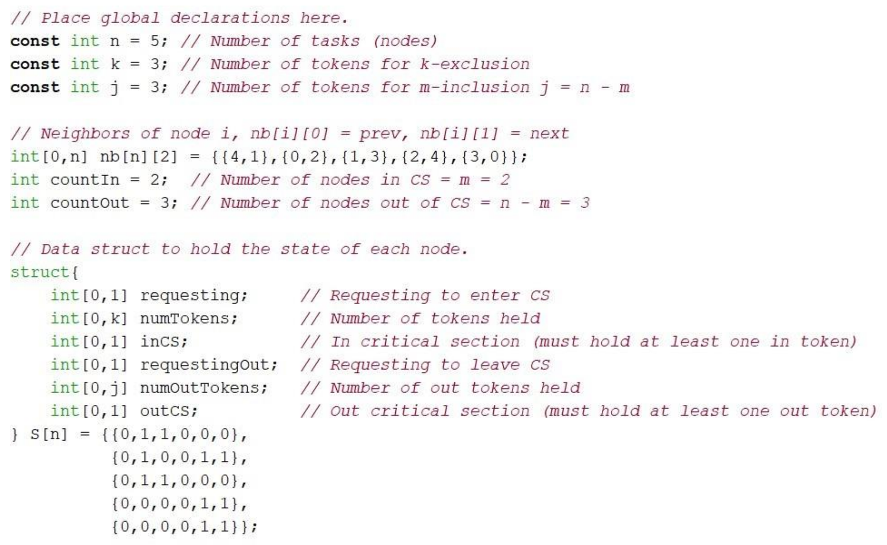
```
E<> (N2.CS && N1.CS)
A[] (count <= j)
A[] (not Obs1.BAD)
E<> (N2.NonCS && S[2].numTokens == 1)
A[] (not deadlock)
```

Figure 7: ETL for Verification in UPPAAL

3.3 *(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 8 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 9, 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.



```
// Place global declarations here.
const int n = 5; // Number of tasks (nodes)
const int k = 3; // Number of tokens for k-exclusion
const int j = 3; // Number of tokens for m-inclusion j = n - m

// Neighbors of node i, nb[i][0] = prev, nb[i][1] = next
int[0,n] nb[n][2] = {{4,1},{0,2},{1,3},{2,4},{3,0}};
int countIn = 2;  // Number of nodes in CS = m = 2
int countOut = 3; // Number of nodes out of CS = n - m = 3

// Data struct to hold the state of each node.
struct{
    int[0,1] requesting;     // Requesting to enter CS
    int[0,k] numTokens;      // Number of tokens held
    int[0,1] inCS;           // In critical section (must hold at least one in token)
    int[0,1] requestingOut;  // Requesting to leave CS
    int[0,j] numOutTokens;   // Number of out tokens held
    int[0,1] outCS;          // Out critical section (must hold at least one out token)
} S[n] = {{0,1,1,0,0,0},
          {0,1,0,0,1,1},
          {0,1,1,0,0,0},
          {0,0,0,0,1,1},
          {0,0,0,0,1,1}};
```

Figure 8: Model Description for *(3, 2)-Exclusion, Inclusion* with Five Nodes

3.3.1 Verification of Properties

The ETL for *(k, m)-Exclusion, Inclusion* is shown in Figure 10 and the justification for the verification of each of the properties is as described below.

1. E<> (N1.CS && N2.CS && N3.CS): Eventually on some path, the nodes N1, N2 and N3 are allowed to enter the CS at the same time.
2. A[] (countIn <= *k*) && (countIn >= *m*): Always on all paths, a minimum of *m* nodes and a maximum of *k* nodes are present in the CS.
3. E<> (countIn > *k*): Eventually on some path, more than *k* nodes are present in the CS. The property is not satisfied as verified by the verifier.
4. A[] (countIn + countOut == 5): Always on all paths, a node is either in CS or outside of CS but never in an intermediary state. Hence, the sum of countIn and countOut equals the number of nodes in the system.
5. A[] (not deadlock): Always on all paths, there are no deadlocks in the system.

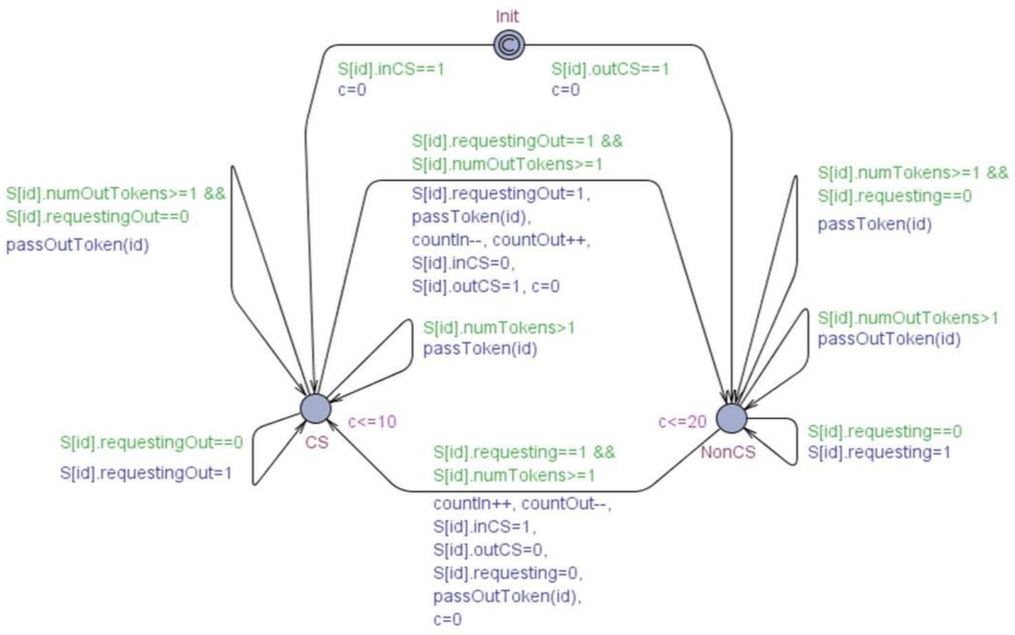
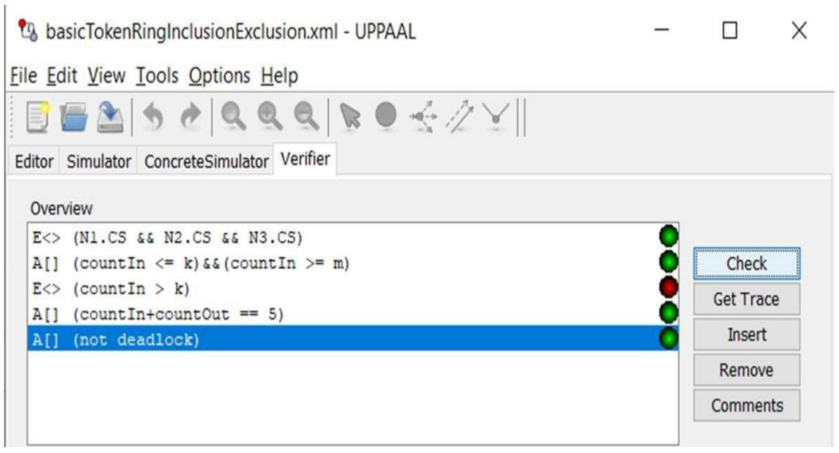


Figure 9: UPPAAL Model for *(k, m)-Exclusion, Inclusion*


basicTokenRingInclusionExclusion.xml - UPPAAL
File Edit View Tools Options Help
Editor Simulator ConcreteSimulator Verifier
Overview
Check
Get Trace
Insert
Remove
Comments

```
E<> (N1.CS && N2.CS && N3.CS)
A[] (countIn <= k)&&(countIn >= m)
E<> (countIn > k)
A[] (countIn+countOut == 5)
A[] (not deadlock)
```

Figure 10: ETL for Verification in UPPAAL

4 Future Work and 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. As part of future work, a self-stabilizing approach for k-exclusion using logical rings and byte tokens is in the works. 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 collaborate 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)