Boarding Pass Issuance for Passengers at Airport

ENSE 623 Systems Engineering

Submitted by

Bargava R Subramanian

Soe Zarni

ABSTRACT

Each airport has a fixed space which it allots to different airlines so that they can set-up their boarding pass issuance counter. We consider a simplistic case where two airlines have decided amongst themselves to one of the counters. The number of counters each airline will operate will depend upon number of passengers arriving for each of the airlines. Given this the boundary of operation of each airlines vary across time. For each airline, given the limited operating space, they have to come up with a queue length for the passengers (whether to have a single line or in some zigzag format).

For this problem statement our approach this semester is divided into two parts

- How do the airlines decide their operating space (spatial constraint)
- The second one is to evaluate the temporal constraints.

The temporal constraint determines

- Whether the passenger who joins the queue is able to catch his flight or not.
- Whether the airline is able to process the passenger within the given time frame so that he is able to catch the flight.

The temporal logic is basically a timed automaton and we are using UPPAAL to verify/validate our system. We will construct the system in UPPAAL using defined requirements and try to verify/validate the system with different queuing time, process time and WIP time calculated from different arrival rates, number of counters, process rate, etc. We will also use Binary Space Paritioning tree to define the spatial constraint and then try to validate/verify it using Spatial Logia Model Checker.

INTRODUCTION

Offering services of the highest quality is of utmost priority for airlines. This decides to a major extent the demand of passengers they obtain. Of course, the cost of tickets too play a major factor – but to determine an optimal cost for the tickets, the airlines need to determine the operating cost and how to optimize them. Given fixed resources, they need to come up with excellent quality. Considering the vast amount of operations they do, we have considered a particular operation – boarding pass issuance and have developed a systems engineering perspective of the same.

The airport has a fixed space for the airlines to set up their counters for their operations. The airlines bid for the slots and they are given a fixed number of counters – over a long-term lease. While having large number of counters will be good when demand is high, not at all times are this feasible – for they cost a lot for the airlines. Generally, airlines share the counters on a mutual agreement. A particular airline might service winter destinations and might have lot of demand in that time. Another airline might offer service to summer destinations and hence their summer demand will be high. It is mutually beneficial for both of them to share counters so that over the course of the year, their operating costs are relative to the service they offer.

With this background, we have performed a study on how do we come up with a systems engineering design for the following problem:

Two partner-airlines have in together a fixed number of counters for their operation. How do they mutually decide on the number of counters that they need to operate at a given time of the day?

When the counters are decided, they need to set up their queue length for the passengers to come and stand in line. The queue boundary is a mobile one. They can be adjusted knowing the demand expected. The airlines know the demand they are going to expect at particular time. Hence, they can decide whether the passengers need to stand in a twirling

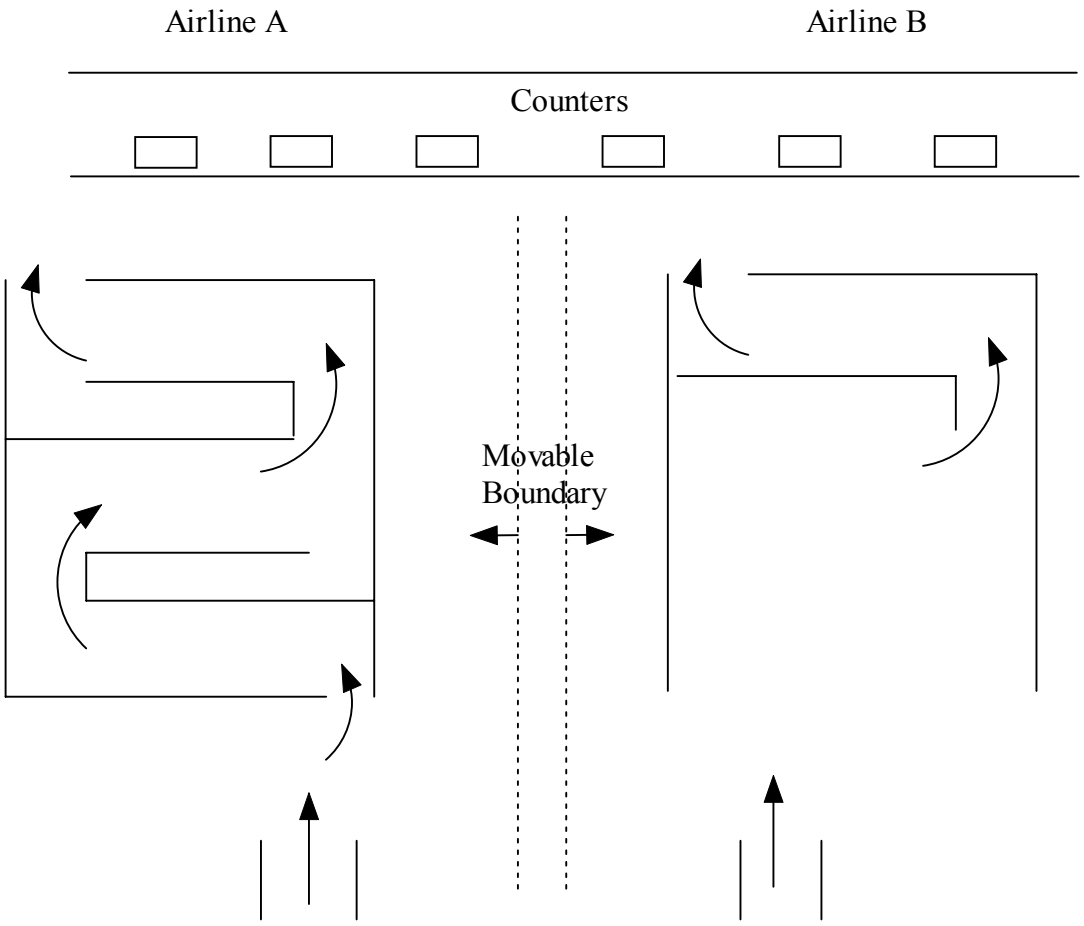
path (like a maze) or come directly to the counters in a single line. The number of bends in the line is a decision issue. When there is a huge rush, the number of bends is high and the bends are very low when the rush is low.

When passengers arrive later than the prescribed boarding time for a particular flight – say 15 min before the departure of the flight, the airlines need to process the passenger immediately. Otherwise, it will cost for the airlines to reaccomodate him in a later flight. Hence, 30 min before the departure of the flight, announcements are made every 5 minutes requesting passengers departing for that flight to come forward (basically jump over the queue). They are processed in the first available counter. Also, there are a lot of people require special services – eg., elderly people, sick people, etc. Hence, they are also processed as and when they come. The implementation of special services offered differs from airline to airline. The issue of how airlines model such behaviors is studied here.

Hence, our project deals with the following objectives:

1) How do two airlines share the counters? How many counters are operated by each of them at a given point of time?
2) What is the queue length set at a given point of time for passengers approaching the counters?

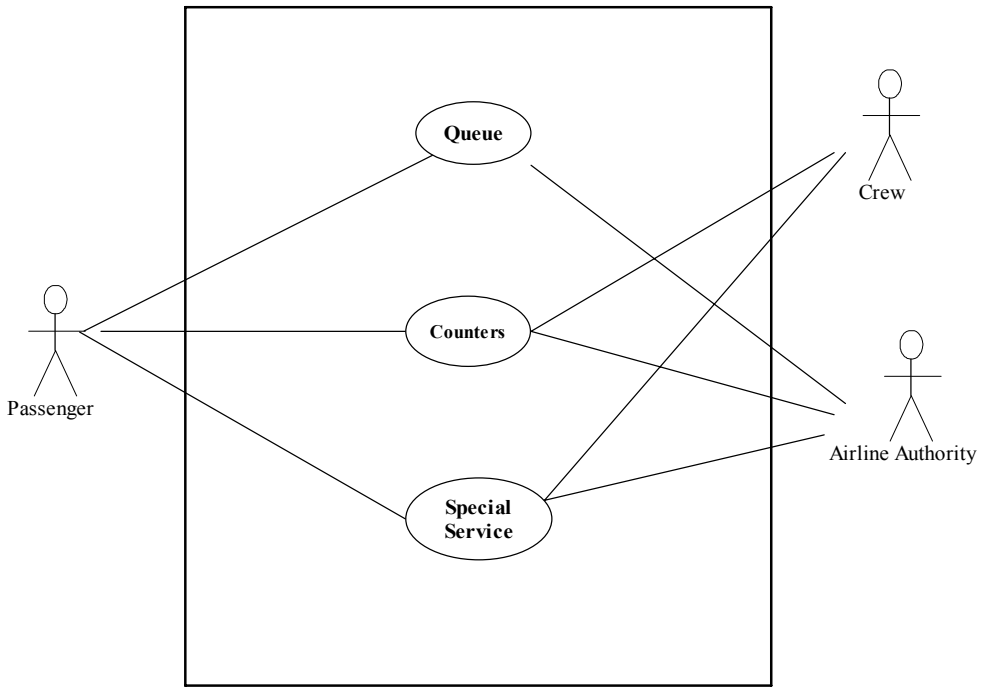
The following diagram represents a typical airport set-up where two airlines are placed next to each other. They share the counters. A mobile boundary exists between them – but the total space is limited. The queue length for each airline needs to be determined. Also, handling of special services is to be decided by the airlines.



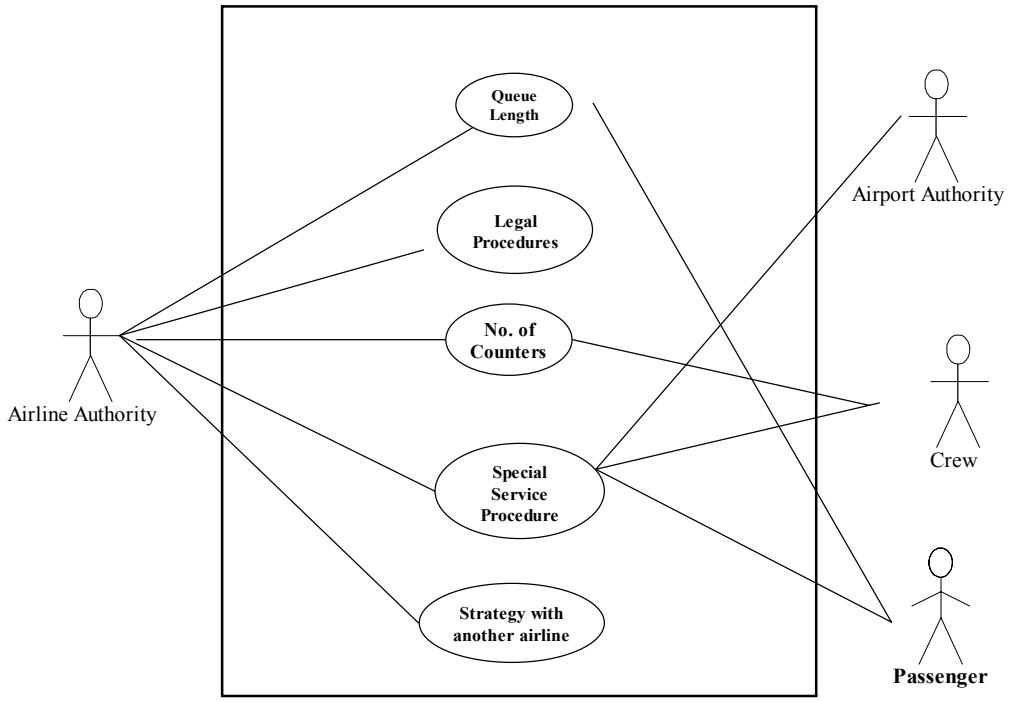
Typical Boarding Pass issuing-counter Setup

Initial Use Case Modeling and Activity Diagram

Initial Use Case Diagram



Use Case Diagram – Passenger Perspective



Use Case Diagram – Airline Perspective

Use Case A. Boarding pass issuance

Description: The passenger obtains the boarding pass – from passenger's perspective

Primary Actors: Passenger, Crew

Use Case B. Determination of Counters/Queue Length

Description: Airlines need to determine the number of counters and the optimal queue length which the passengers have to traverse in order to get the boarding pass – From Airlines perspective

Primary Actors: Airlines, Crew, Partner-Airline, Passenger, Airport Authorities

Pre-Conditions:

- Airport authorities to airlines allocate fixed number of counters.
- Two airlines share the counters.
- Passenger demand at a given point of time is known.

Flow of Events:

B-1. Passengers arrive at the correct landing point per their schedule

B-2. The demand at a given time is known. Hence the queue length is set.

B-3. The airline and its partner airline mutually decide to share counters as per their demand at a given point of time.

B-4. Adequate crew members are allocated.

B-5. Process for special services-required passengers is determined.

Post Conditions:

- Passenger boards plane or compensated appropriately.

Alternate Flow of Events: None

Assumptions: None

Goals and Scenarios

Goal 1. Proper queuing mechanism should be setup for passengers arriving on time.

Scenario 1.1. Arriving passengers wait at the lounge till their flight announcement is made.

Scenario 1.2. Once announcement is made, passengers join at the end of the queue that leads to the counters for issuance of boarding pass.

Scenario 1.3. There is just a single queue. Once a particular counter is free, the next passenger in line gets serviced.

Goal 2. Separate Queuing Mechanism for passengers arriving late

Scenario 2.1. Passengers might typically come later than the initial check-in time suggested. (Eg: 15 min before flight take-off). 20 min before the actual flight take-off, an announcement is made to determine the passengers who aren't yet processed for that flight. Those passengers get processed immediately by the service team.

Scenario 2.2. Care is taken as not to allocate all the counters to such late-passengers as it penalizes people arriving on time for the next flight.

Goal 3. Flight Announcement should be made at proper time

Scenario 3.1. Passengers waiting at the lounge should be given a first call-for picking up ticket issuance 90 minutes before flight taken. After that every 30 min a call is given and when just 20 min is left for take-off, a separate mechanism is activated.

Scenario 3.2. The passengers should join the queue.

Goal 4. The service-team size should be of optimal size

Scenario 4.1. The service team shouldn't be either in shortage or in excess. Proper trade-off is carried out to determine actual team size at a given time interval.

Scenario 4.2. Breaks and change of team members should be done smoothly.

Goal 5. Mechanism for passengers missing flight

Scenario 5.1. If passenger arrives after flight has taken-off, he has to buy another ticket.

Scenario 5.2. If passenger misses flight because of either connecting flight delay or because of the delay caused by the service team, adequate measures are taken to compensate the passenger and to ensure that he takes next available flight.

Goal 6. The efficiency and effectiveness of the system should be high

Scenario 6.1. The quality and cost of service provided should be at optimal.

Scenario 6.2. Periodic assessments of the various parameters are to be carried out to ensure that efficiency and effectives are maintained at the highest level.

Scenario 6.3. The system should be highly reliable and feasible.

Scenario 6.4. The mean service rate should be optimal considering the mean arrival rate of passengers.

Goal 8. The service team should have support personnel

Scenario 8.1: There should be enough support personnel to take care of unprecedented events.

Scenario 8.2: There should be a maintenance Training/Support Team

Goal 9. The system should be capable to withstand active and/or rigorous usage.

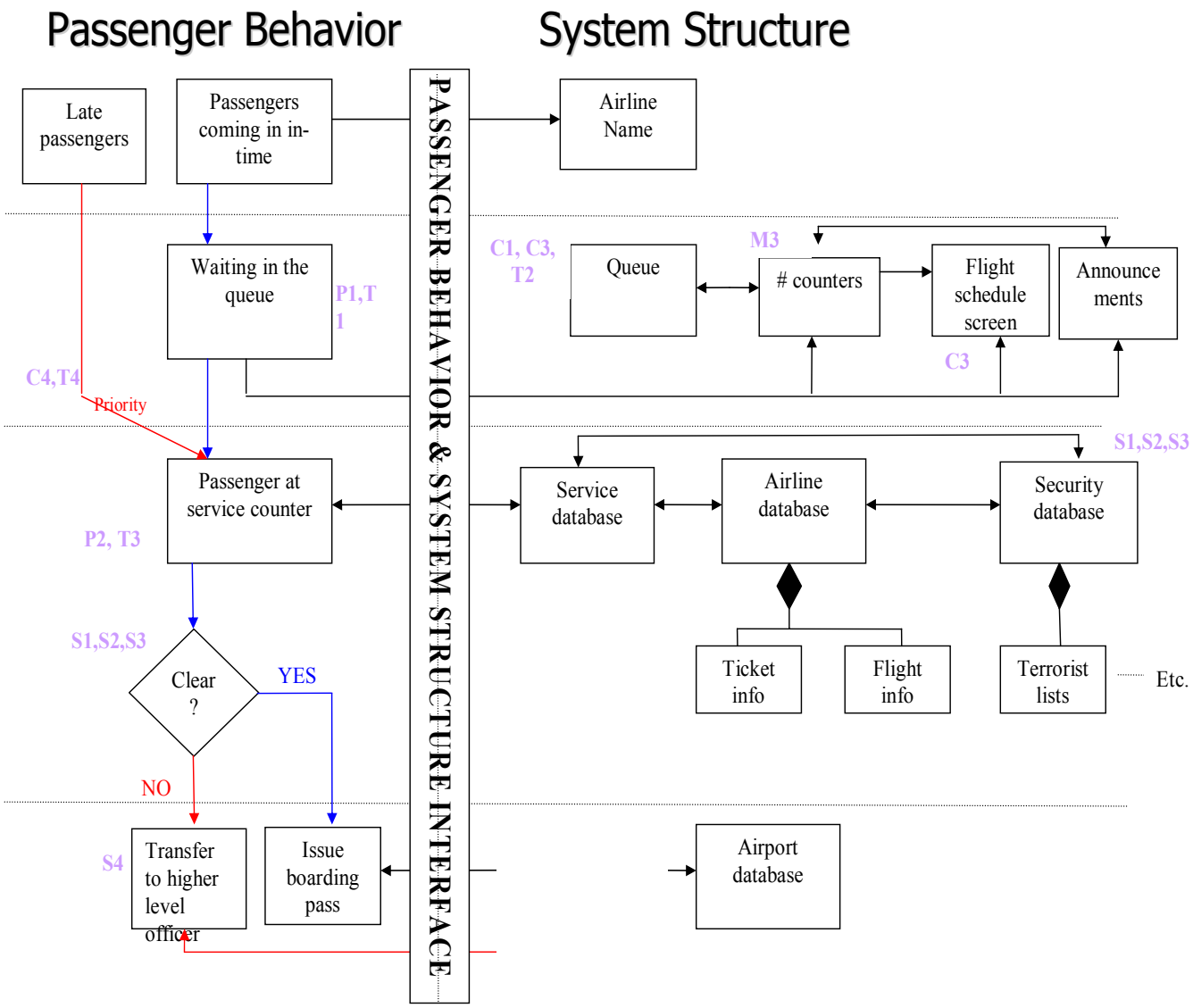
Scenario 9.1. Even when the number of passengers arriving at a particular time exceeds the normal arrival rate, the operations should be carried out efficiently.

Requirements Traceability Matrix

	Requirements	Goal	Scenario
	Management Requirements		
M1	The system must be reliable.	8,9	8.1,9.1
M2	The cost of the system must be minimum.	6	6.1
M3	The utilization of each employee must be at least 80%.	4,6,8	4.1,8.1,8.2
	Airline Requirements		
A1	The passengers must be able to check-in in time for the flights.	2,3	2.1,3.1
A2	The system must be able to handle the flight schedule changes.	3	3.1,3.2
A3	The system must make sure all the passengers already issued boarding passes are cleared for security from all database.	6	6.1
	Crew Counters Requirements		
C1	The system must be able to handle any queue size.	4,8,9	4.1,4.2,8.1, 9.1
C2	The system should have at least a supervisor available at all time to solve problems and make decision.	6	6.1,6.2,6.3, 6.4
C3	The system must be able to handle late passengers and flight schedule changes.	5	5.1,5.2
C4	The late passengers must have priority to get the service.	2	2.1
	Passengers Requirements		
P1	The wait time in the queue must be as short as possible.	6	6.3,6.4
P2	The processing time for check-in time must be as quick as possible.	6	6.3
P3	The passengers must be able to board on the plane in time.	2,3	2.1,3.1
P4	There should be an alternative way if passenger missed the flight.	5	5.1,5.2
	Timing Requirements		
T1	The waiting time of any passenger in the queue must be less than 15 mins.		
T2	The number of passengers in the queue at any time must be less than 10.		
T3	The service time of any passenger must be less than 10 mins.		
T4	Any late passenger must get service in less than 3 mins of waiting time.		

In the current Project, we do no consider security arrangements while modeling and verifying and validating the state and structure of the behavior.

Mapping of System structure and behavior



The above diagram shows the queuing system structure and behavior and how they are mapped at each stage of passenger action. When a passenger walks into the lounge, he has to look for the airline name first. So the interface for communication between behavior and system is the airline name.

Now the passenger is waiting for the service in the queue which needs to conform to requirements P1 and T1. On the structure side, there are queue with requirements C1, C3 and T2, number of counters (M3), Flight schedule screen (C3) and Announcements. The counters have to interact with flight schedule, queue and announcements in order to know the departing flight, adjust the number of counters and to be able to serve the late passengers in time. The interfaces for communication between behavior and structure are number of counters, flight schedule and announcements.

Now the passenger is at the service counter (P2 and T3). But the late passengers have priority to get the service according to requirements C4 and T4. Here the crew needs to check the service data base for the passenger information, airline data base for the ticket and flight information, and security database (S1,S2 and S3) for the security clearance.

If everything is cleared, the crew interacts with airport database and issues the boarding pass to the passenger. If not cleared, the passenger is transferred the supervisor according to requirement S4.

Concurrent System – Analysis :

The system has the following concurrent aspects:

1) The queue length changes dynamically.
2) The number of counters used by the airline and its partner varies at a given point of time.
3) Passengers can be late or might require special service.

The system has to be subject to both temporal and spatial analysis to determine about implied scenario and to check the model for its validity. Also, since this is a concurrent system, we need to ensure that deadlocks don't occur.

Further scope is to improve upon the basic model by implementing all the functionalities and to include timing constraints as well.

The analysis has to perform the following :

Solving Spatial Constraint

- How do the airlines determine their operating space - one or multiple counters ?
- How should their queue structure be – single line or multiple lines ?

Solving Temporal Constraint

- Do the passengers catch the flight on time ?
- Do the airlines process the passengers so that they catch the flight ?

  - Yes – but there is an assumption – to avoid complications – we assume that if airline obtain extra counter – they process the extra burden and are well-off.

Necessity for Spatial Analysis

There are several key factors that force us to do spatial analysis of the system.

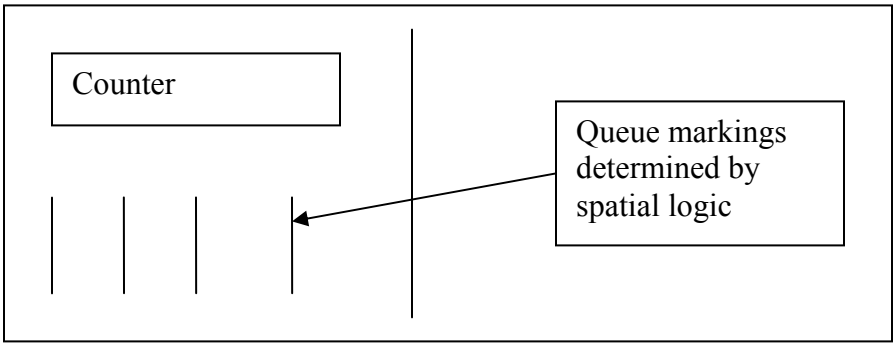
1. To explore limited space - difficulties with components sharing a single resource. Requirements are to be developed to prevent this type of resource misuse.
2. To identify scenarios which aide in the changing of the spatial occupation of resources. In a nutshell, to perform the task of identifying all potential system paths based on our system definition that change spatial structure.
3. Deadlocks are to be identified in the system specification and corrected – deadlock in space means two resources occupying same space.
4. Space is a key performance factor.
5. Space is required to perform the function of a safety factor.
6. Availability of resources are determined by space.
7. For efficiency, resources must perform their functions in a set amount of space because of interaction with other components. Waiting time for space needs to be minimized.
8. The space of operation of a function is performed may determine the output.
9. Multiplicity of class related objects, how do space requirements change when there are multiple of the same type of object.
10. The order in which actions take place or states are reached is related to proper operation and utilization of space.

Spatial Constraint :

Accomplish the following tasks :

- Determine number of operating counters required at a specific time
- Given operating space for the airlines, the queue length has to be determined.

For the boarding pass issuance, the counter's queue length has to be split.



Procedure :

1) Start each airline with just one counter and with just one queue line
2) Check periodically the population of the queue.
3) Whenever the queue becomes congested, add one more line
4) When the whole counter space is nearing capacity, request for the next counter.
5) Temporal logic provides solution of when next counter is made available.
6) Reassign population amongst both the counters in a fair manner.
7) Release counter when done or when requested.

BINARY SPACE PARTITIONING TREE

- We obtain the breaking of queue into minuscule queues by BSP tree.
- Binary space partitioning (BSP) is a method for recursively subdividing a space into convex sets by hyperplanes.
- This subdivision gives rise to a representation of the scene by means of a tree data structure known as a BSP tree.

"BSP tree construction is a process which takes a subspace and partitions it by any hyperplane that intersects the interior of that subspace. The result is two new subspaces that can be further partitioned by recursive application of the method. "

- Our problem is a BSP tree data-structure - variant.
- A point is represented by a vector P(x,y,z)
- A plane is represented by a vector (P,p). P is the normal vector to plane.

Possible Extensions : Incorporate Quad-trees for queue formation. Quad-trees recursively divide given subspace into 4 subspaces.

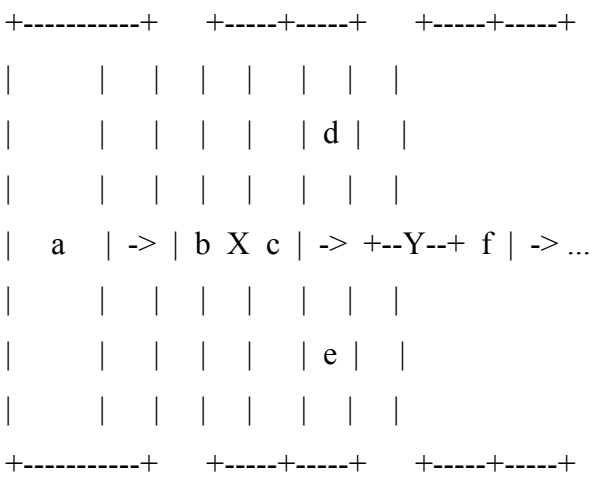
What is a BSP Tree?

A Binary Space Partitioning (BSP) tree represents a recursive, hierarchical partitioning, or subdivision, of n-dimensional space into convex subspaces. BSP tree construction is a process which takes a subspace and partitions it by any hyperplane that intersects the interior of that subspace. The result is two new subspaces that can be further partitioned by recursive application of the method.

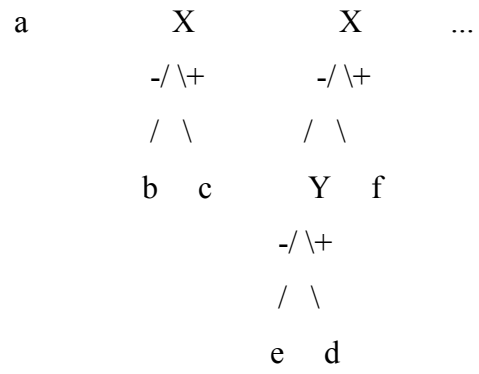
A "hyperplane" in n-dimensional space is an n-1 dimensional object which can be used to divide the space into two half-spaces. For example, in three dimensional space, the "hyperplane" is a plane. In two dimensional space, a line is used.

BSP trees are extremely versatile, because they are powerful sorting and classification structures. They have uses ranging from hidden surface removal and ray tracing hierarchies to solid modeling and robot motion planning.

We use only lines parallel to the X or Y axis, and that we will divide the space equally at each node. For example, given a square somewhere in the XY plane, we select the first split, and thus the root of the BSP Tree, to cut the square in half in the X direction. At each slice, we will choose a line of the opposite orientation from the last one, so the second slice will divide each of the new pieces in the Y direction. This process will continue recursively until we reach a stopping point, and looks like this:



The resulting BSP tree looks like this at each step:



Building BSP Trees :

Given a set of polygons in three dimensional space, we want to build a BSP tree which contains all of the polygons. For now, we will ignore the question of how the resulting tree is going to be used.

The algorithm to build a BSP tree is very simple:

1. Select a partition plane.
2. Partition the set of polygons with the plane.
3. Recurse with each of the two new sets.

Choosing the partition plane

The choice of partition plane depends on how the tree will be used, and what sort of efficiency criteria you have for the construction. For some purposes, it is appropriate to choose the partition plane from the input set of polygons. Other applications may benefit more from axis aligned orthogonal partitions.

In any case, you want to evaluate how your choice will affect the results. It is desirable to have a balanced tree, where each leaf contains roughly the same number of polygons. However, there is some cost in achieving this. If a polygon happens to span the partition plane, it will be split into two or more pieces. A poor choice of the partition plane can result in many such splits, and a marked increase in the number of polygons. Usually there will be some trade off between a well balanced tree and a large number of splits.

Partitioning polygons

Partitioning a set of polygons with a plane is done by classifying each member of the set with respect to the plane. If a polygon lies entirely to one side or the other of the plane, then it is not modified, and is added to the partition set for the side that it is on. If a polygon spans the plane, it is split into two or more pieces and the resulting parts are added to the sets associated with either side as appropriate.

When to stop

The decision to terminate tree construction is, again, a matter of the specific application. Some methods terminate when the number of polygons in a leaf node is below a maximum value. Other methods continue until every polygon is placed in an internal node. Another criteria is a maximum tree depth.

C++ CODE

The algorithm(in C++ style) of the C++ code is as follows :

```
struct  BSP_tree
{
            plane     partition;
                      list      polygons;
            BSP_tree  *front,
            *back;
};
```

It stores pointers to its children, the partitioning plane for the node, and a list of polygons coincident with the partition plane.

```
void    Build_BSP_Tree (BSP_tree *tree, list polygons)
{
  polygon   *root = polygons.Get_From_List ();
  tree->partition = root->Get_Plane ();
  tree->polygons.Add_To_List (root);
  list      front_list,
            back_list;
  polygon   *poly;
  while ((poly = polygons.Get_From_List ()) != 0)
```

```
  {
    int   result = tree->partition.Classify_Polygon (poly);
    switch (result)
    {
      case COINCIDENT:
        tree->polygons.Add_To_List (poly);
        break;
      case IN_BACK_OF:
        backlist.Add_To_List (poly);
        break;
      case IN_FRONT_OF:
        frontlist.Add_To_List (poly);
        break;
      case SPANNING:
        polygon   *front_piece, *back_piece;
        Split_Polygon (poly, tree->partition, front_piece, back_piece);
        backlist.Add_To_List (back_piece);
        frontlist.Add_To_List (front_piece);
        break;
    }
  }
  if ( ! front_list.Is_Empty_List ())
  {
    tree->front = new BSP_tree;
    Build_BSP_Tree (tree->front, front_list);
  }
  if ( ! back_list.Is_Empty_List ())
  {
    tree->back = new BSP_tree;
    Build_BSP_Tree (tree->back, back_list);
  }
}
```

This routine recursively constructs a BSP tree using the above definition. It takes the first polygon from the input list and uses it to partition the remainder of the set. The routine then calls itself recursively with each of the two partitions. This implementation assumes that all of the input polygons are convex.

Classifying a point with respect to a plane is done by passing the (x, y, z) values of the point into the plane equation, $Ax + By + Cz + D = 0$. The result of this operation is the distance from the plane to the point along the plane's normal

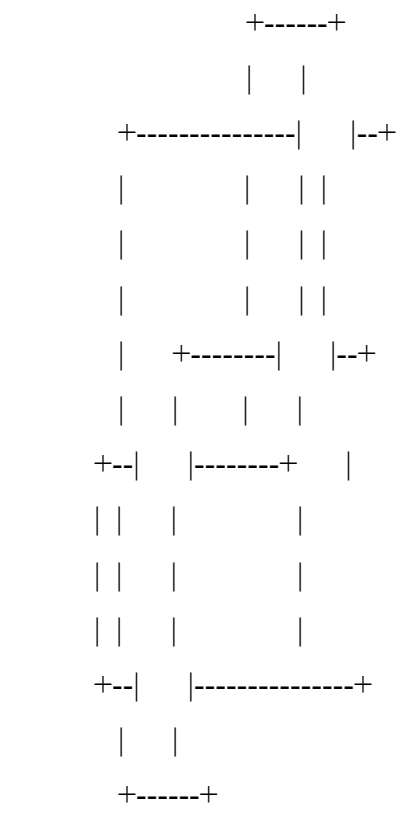
vector. It will be positive if the point is on the side of the plane pointed to by the normal vector, negative otherwise. If the result is 0, the point is on the plane.

Here is a very basic function to split a convex polygon with a plane:

```
void Split_Polygon (polygon *poly, plane *part, polygon *&front, polygon
*&back
)
{
  int   count = poly->NumVertices (),
        out_c = 0, in_c = 0;
  point ptA, ptB,
        outpts[MAXPTS],
        inpts[MAXPTS];
  real sideA, sideB;
  ptA = poly->Vertex (count - 1);
  sideA = part->Classify_Point (ptA);
  for (short i = -1; ++i < count;)
  {
    ptB = poly->Vertex (i);
    sideB = part->Classify_Point (ptB);
    if (sideB > 0)
    {
      if (sideA < 0)
      {
        // compute the intersection point of the line
        // from point A to point B with the partition
        // plane. This is a simple ray-plane intersection.
        vector v = ptB - ptA;
        real   sect = - part->Classify_Point (ptA) / (part->Normal () | v);
        outpts[out_c++] = inpts[in_c++] = ptA + (v * sect);
      }
      outpts[out_c++] = ptB;
    }
    else if (sideB < 0)
    {
      if (sideA > 0)
      {
        // compute the intersection point of the line
        // from point A to point B with the partition
        // plane. This is a simple ray-plane intersection.
```

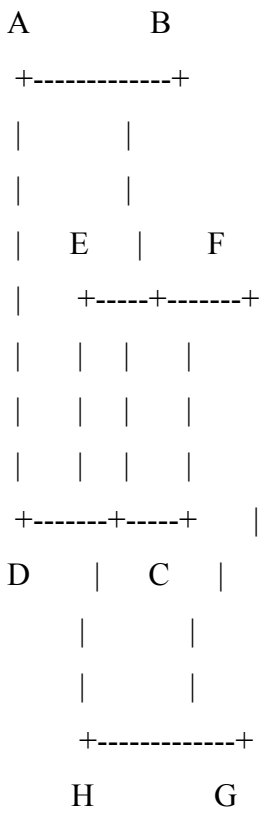
```
            vector v = ptB - ptA;
            real   sect = - part->Classify_Point (ptA) / (part->Normal () | v);
            outpts[out_c++] = inpts[in_c++] = ptA + (v * sect);
          }
          inpts[in_c++] = ptB;
        }
        else
          outpts[out_c++] = inpts[in_c++] = ptB;
        ptA = ptB;
        sideA = sideB;
      }
      front = new polygon (outpts, out_c);
      back = new polygon (inpts, in_c);
    }
```

The output will be something like this :

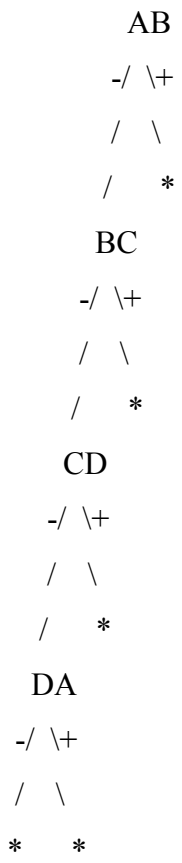


Draw the subtree at the far child from the eye, then draw the polygons in this node, then draw the near subtree. Repeat this procedure recursively for each subtree.

It is useful to examine the construction process in two dimensions.

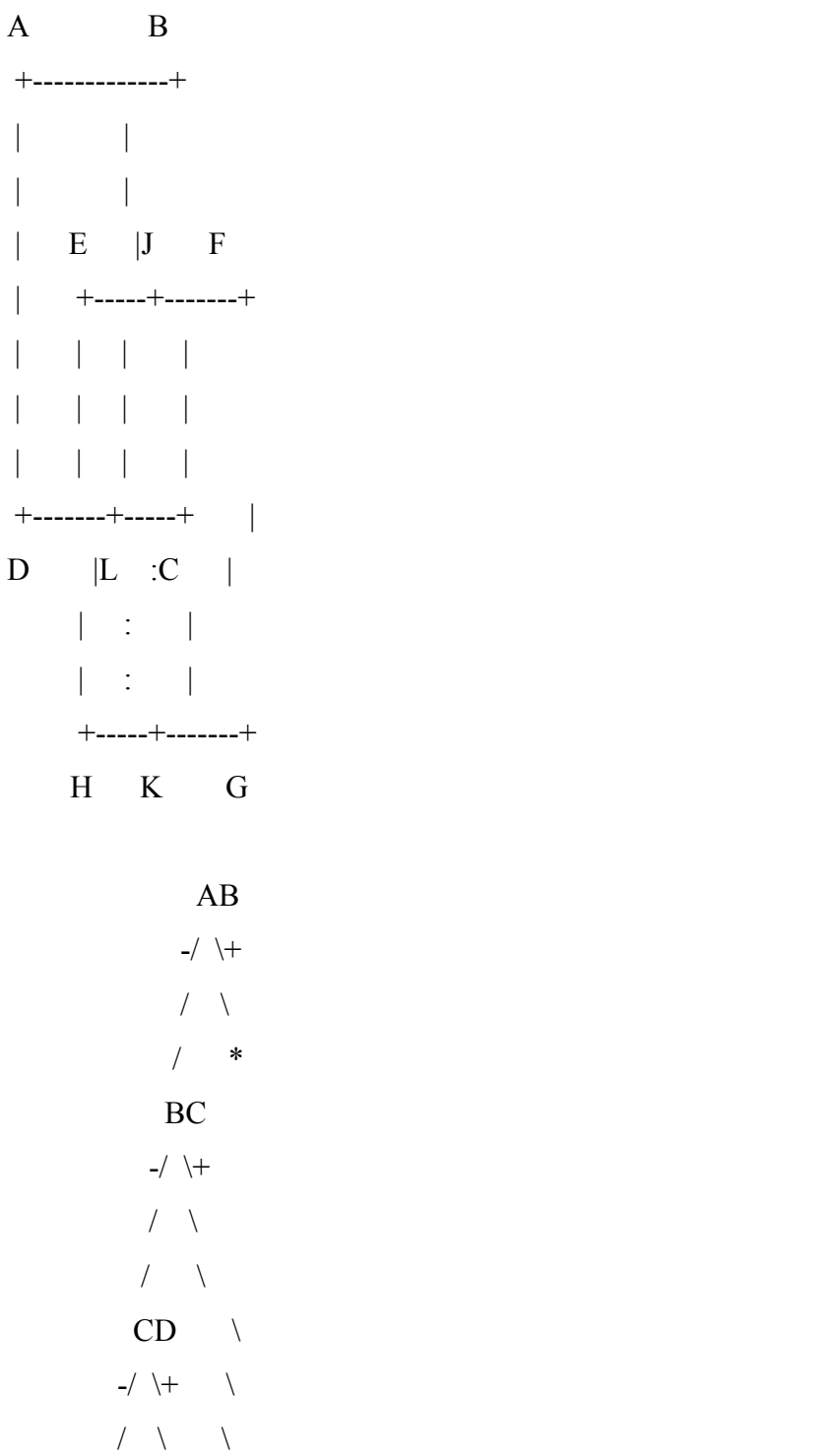


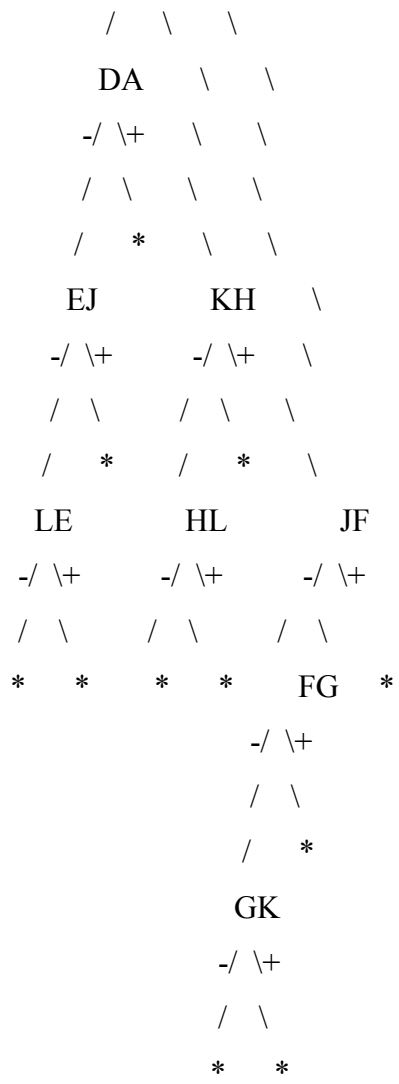
Two polygons, ABCD, and EFGH, are to be inserted into the tree. We wish to find the union of these two polygons. Start by inserting polygon ABCD into the tree, choosing the splitting hyperplanes to be coincident with the edges. The tree looks like this after insertion of ABCD:



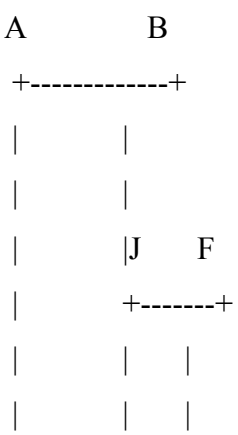
Now, polygon EFGH is inserted into the tree, one polygon at a time.

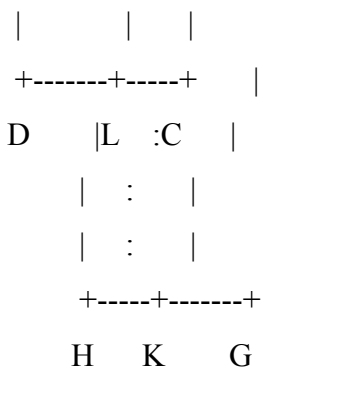
The result looks like this:



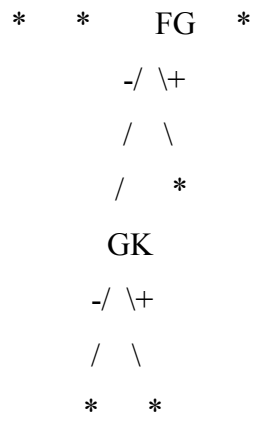


Our tree now looks like this:




D
L
C
H
K
G


AB
BC
CD
DA
KH
HL
JF
*
*
*
*



Thus effectively, the space is split into the required parts using BSP.

Each time the queue reaches its maximum amount – the space is split so that one more line is formed. When max. tree depth is reached – a request for new counter is made – which is taken care by temporal constraints.

Modeling,Verifying and Evaluating Spatial Logic

Spatial logics support the specification not only of behavioral properties but also of structural properties of concurrent systems, in a fairly integrated way. Spatial properties arise naturally in the specification of distributed systems. In fact, many interesting properties of distributed systems are inherently spatial, for instance connectivity, stating that there is always an access route between two different sites, unique handling, stating that there is at most one server process listening on a given channel name, or resource availability, stating that a bound exists on the number of channels that can be allocated at a given location. Secrecy can also be sometimes understood in spatial terms, since a secret is a piece of data whose knowledge of is restricted to some parts of a system, and unforgettable by other parts. Spatial logics have been used in the definition of several core languages, calculi, and data models.

The ***Spatial Logic Model Checker*** is a tool allowing the user to automatically verify behavioral and spatial properties of distributed and concurrent systems expressed in a pi-calculus. The algorithm implemented (currently using on-the-fly model-checking techniques) is provably correct for all processes, and complete for the class of bounded processes. This uses the language – pi-calculus.

We attempted at evaluating this spatial logic checker and we tried to model this :

- Define the system.
- Model the fixed space (defined by specific boundaries)
- Specify spatial logic
- Spatial logic is defined by having to split the space in a particular counter into two for multiple queues.
- The counter is assumed to be full when maximum threshold for the split has occurred (pre-defined).
- Modeled it based on the examples provided in the manual (the language pi-calculus is a major restriction in understanding and implementing the system)

SPATIAL LOGIC MODEL CHECKER/VERIFIER FOR COUNTER-QUEUES

```
/* SYSTEM definition */

defproc PaxEntry(inRt,outRt) =  select {[outRt=inRt].0; outRt!(inRt).0};

defproc CounterCount(inRt,outRt) =
        inRt?(newInRt).CounterOwner(newInRt,outRt)
        and CounterOwner(inRt,outRt) =

select
 {
        tau.Exit(inRt,outRt);
        outRt!(outRt).CounterCount(inRt,outRt)};

        defproc System =
        (new l1,l2,l3,l4,l5 in
        (CounterCount(l1,l2) |
        CounterCount(l2,l3) |
        CounterCount(l3,l4) |
        CounterCount(l4,l5) |
        CounterOwner(l5,l1)));
}

/* SYSTEM PROPERTIES */

check System |= always eventually 0;

/***/

defprop Leave(inl,outl) = 1
and
(((
     inl != outl) or <>0)
      and <outl!(inl)> 0);

     defprop count(inl,outl) = 1
    and
        (Leave(inl,outl) or
        (maxfix X(inLnk).
        ((<inLnk?(newInLnk)> X(newInLnk))
        or ((<> Leave(inLnk,outl))
        and (<outl!(outl)> X(inLnk)))))
        (inl)
);

defprop Leave(inl,outl) =1 and
```

```
(((
        inl != outl) or <>0)
        and <outl!(inl)> 0);
        defprop count(inl,outl) =
        1 and
        (Leave(inl,outl) or
        (maxfix X(inLnk).
        ((<inLnk?(newInLnk)> X(newInLnk))
        or ((<> Leave(inLnk,outl))
         (inl)
);


TerminalOwner(find,Shift,obj) =
find?(changeCntr,cntrRequest).Owner(find,Shift,cntrRequest,changeCntr,obj)
and
Owner(find,Shift,link,queue,obj) =
select
{
        find?(changeCntr,cntrRequest).(Owner(find,Shift,cntrRequest,queue,obj)
        | link!(changeCntr,find));
        tau.(Idle(find,Shift,link) | queue!(obj))
}
and
Idle(find,Shift,link) = select
{
        find?(changeCntr,cntrRequest).(Idle(find,Shift,cntrRequest)
        | link!(changeCntr,find));
        tau.(TerminalCounterChange(find,Shift) | link!(Shift,find))
}
and
TerminalCounterChange(find,Shift) =
select
{
      find?(changeCntr,cntrRequest).CounterChange(find,Shift,cntrRequest,changeCntr);
      Shift?(obj).TerminalOwner(find,Shift,obj)
}
and
CounterChange(find,Shift,link,queue) =
select
{
        find?(changeCntr,cntrRequest).(CounterChange(find,Shift,cntrRequest,queue)
        | link!(changeCntr,find));
        Shift?(obj).Owner(find,Shift,link,queue,obj)
}
;
/* --- */
/* MORE  PROPERTIES  - The system should be deadlock free */

defprop deadlockfree = always(<>true);
check Dir |= deadlockfree;
/* ---------- */
defprop object(s) = <s!>0;
```

```
defprop count(f) = 1 and (fresh a. fresh b. <f?(a,b)>true);
defprop owns(i,obj) = (count(i) and @obj);
defprop exclusive(i,obj) = (owns(i,obj) | not @obj);
eventually exclusive(i,obj)));
check Dir |= always(live);
```

Necessity for Temporal Analysis

There are several key factors that force us to do temporal analysis of the system.

- To analyze concurrency - difficulties with components sharing a single resource. Requirements are to be developed to prevent this type of resource misuse.
- To identify implied scenarios which helped close down missing requirements in the system. In a nutshell, to perform the task of identifying all potential system paths based on our system definition.
- Deadlocks are to be identified in the system specification and corrected.
- Time is a key performance factor.
- Time is required to perform the function of a safety factor.
- Availability of resources is determined by time.
- For efficiency, resources must perform their functions in a set amount of time because of interaction with other components. Waiting time needs to be minimized.
- The rate at which a function is performed may determine the output.
- Multiplicity of class related objects, how do timing requirements change when there are multiple of the same type of object.
- The order in which actions take place or states are reached is related to proper operation.

UPPAAL AND TEMPORAL LOGIC

- Simple system may be drawn out and analyzed by inspection, but larger systems with multiple states and objects require a better method.
- Instead of verifying single “test cases” with traditional verification methods, UPPAAL allows testing of all paths across a range of times.

- Multiple instantiations are allowed of a single class of objects with different input parameters.

- Local time requirements may be applied to states and transitions to denote lower level requirements on time.
- Global or high level time requirements may be tracked with a global clock that updates according to local time steps.
- Temporal requirements may be verified against the system model by using temporal logic queries.
- A visual representation of object states is given that conserves space.
- The visual representation also allows validation of system processes.

Time in Boarding Pass issual :

Temporal effects in the system are most easily seen by the time related requirements in several of the key objects.

A. Airline:

- The counters have to be available on time for their operation to proceed smoothly
- They should process the passengers before their flight takes off.

B. Passenger:

- They should be able to catch the flight if they arrive before their flight arrival time.
- They should have ample space in the queue for free movement (provided by the fact that the airline is able to obtain extra counter subject to temporal constraints)

UPPAAL

Uppaal is a tool box for validation (via graphical simulation) and verification (via automatic model-checking) of real-time systems. It consists of two main parts: a graphical user interface and a model-checker engine. The user interface is implemented in Java and is executed on the users work station. It requires that Java 1.2 or higher is

installed on the computer. The engine part is by default executed on the same computer as the user interface, but can also run on a more powerful server. The idea is to model a system using timed-automata, simulate it and then verify properties on it. Timed-automata are finite state machines with time. A system consists of a network of processes that are composed of locations. Transitions between these locations define how the system behaves. The simulation step consists of running interactively the system to check that it works as intended. Then we can ask the verifier to check reachability properties, i.e. if a certain state is reachable or not. This is called model-checking and it is basically an exhaustive search that covers all possible dynamic behaviours of the system. More precisely, the engine uses on-the-y verification combined with a symbolic technique reducing the verification problem to that of solving simple constraint systems . The verifier checks for simple invariants and reachability properties for efficiency reasons. Other properties may be checked by using testing automata or the decorated system with debuggin information

Time in Uppaal

The time model in Uppaal is continuous time. Technically, it is implemented as regions and the states are thus symbolic, which means that at a state we do not have any concrete value for the time, but rather differences. To grasp how the time is handled in Uppaal we will study a simple example. We will use an observer to show the differences. Normally an observer is an add-on automaton in charge of detecting events without perturbing the observed system. In our case the reset of the clock (x:=0) is delegated to the observer to make it work, the original behaviour with the reset directly on the transition loop to itself is not changed actually.

Templates automata are defined with a set of parameters that can be of any type (e.g., int, chan). These parameters are substituted for a given argument in the process declaration.

Constants are declared as const name value. Constants by definition cannot be modified and must have an integer value.

Bounded integer variables are declared as int[min,max] name, where min and max are the lower and upper bound, respectively.
Guards, invariants, and assignments may contain expressions ranging over bounded integer variables. The bounds are checked upon verification and violating a bound leads to an invalid state that is discarded (at run-time). If the bounds are omitted, the default range of -32768 to 32768 is used.

Binary synchronisation channels are declared as chan c. An edge labeled with c! synchronises with another labelled c?. A synchronisation pair is chosen non-deterministically if several combinations are enabled.

Broadcast channels are declared as broadcast chan c. In a broadcast synchronization one sender c! can synchronise with an arbitrary number of receivers c?. Any receiver than can synchronise in the current state must do so. If there are no receivers, then the sender can still execute the c! action, i.e. broadcast sending is never blocking.

Urgent synchronisation channels are decalred by prefixing the channel declaration with the keyword urgent. Delays must not occur if a synchronization transition on an urgent channel is enabled. Edges using urgent channels for synchronisation cannot have time constraints, i.e., no clock guards.

Urgent locations are semantically equivalent to adding an extra clock x, that is reset on all incomming edges, and having an invariant x<=0 on the location. Hence, time is not allowed to pass when the system is in an urgent location. Committed locations are even more restrictive on the execution than urgent locations. A state is committed if any of the locations in the state is committed. A committed state cannot delay and the next transition must involve an outgoing edge of at least one of the committed locations.

Guard :A guard is a particular expression satisfying the following conditions: it is side-e_ect free; it evaluates to a boolean; only clocks, integer variables, and constants are

referenced (or arrays of these types); clocks and clock differences are only compared to integer expressions; guards over clocks are essentially conjunctions (disjunctions are allowed over integer conditions).

Synchronisation : A synchronisation label is either on the form Expression! or Expression? or is an empty label. The expression must be side-effect free, evaluate to a channel, and only refer to integers, constants and channels. Assignment An assignment label is a comma separated list of expressions with a side-effect; expressions must only refer to clocks, integer variables, and constants and only assign integer values to clocks.

Invariant : An invariant is an expression that satisfies the following conditions: it is side-effect free; only clock, integer variables, and constants are referenced; it is a conjunction of conditions of the form x<e or x<=e where x is a clock reference and e evaluates to an integer.

With UPPAAL, we can determine the following states :

Reachability Properties Reachability properties are the simplest form of `properties. They ask whether a given state formula, ', possibly can be satisfied by any reachable state. Another way of stating this is: Does there exist a path starting at the initial state, such that ' is eventually satis_ed along that path.

Safety Properties Safety properties are on the form: “something bad will never happen". In Uppaal these properties are formulated positively, e.g., something good is invariantly true. Let ' be a state formulae. We express that ' should be true in all reachable states with the path formulae.

Liveness Properties Liveness properties are of the form: something will eventually happen.

UPPAAL Introdution

The UPPAAL tool consists of three main parts used for analyzing a system:

1. system editor
2. simulator
3. verifier

System Editor

The system editor is used to input the system description for analysis. A system is described by a set of processes templates, global and local declarations, process assignments, and a system definition. State nodes and transitions are used to represent a system. The select tool is used to select, move, modify and delete these elements. State nodes can be labeled as ‘initial’, ‘urgent’, or ‘committed’.

Simulator

The simulator is a validation tool that permits the visualization of dynamic system behavior during system development. By simulating system behavior with a real-time tool, individual behavior traces may be executed to determine if design goals are reached. Several views are available through the simulator. One window displays available transitions that can be individually selected to walk through a system use-case scenario. When user selected transitions are made another window displays state transitions made by each object in the system. Current states and next-available transitions are highlighted. Global and local variables along with clock values are shown during a system trace and updated consistent with the trace. A sequence diagram view is also available, which shows state progression for individual objects and shared communications ‘synchronization channels’ between objects. Vertical lines denote processes and horizontal lines denote synchronizations.

Verifier

The verifier mode allows simple logical queries to made of the system. These queries allow for the verification of system requirements and determinations of process deadlocks. UPPAAL responds with a comment that "the property is satisfied" or "the property is not satisfied". The language type to interface with the verifier is referred to as requirements specification language.

Several types of verification questions may be used. For instance it is possible to test whether a certain process is in a given location using expressions on the form 'process.location', where 'process' is the process object name and 'location' is the name of the state.

The syntax 'p --> q' represents a 'leads to property' meaning 'A[] (p imply A<> q)'. Essentially, 'p --> q' holds if and only if whenever 'p' holds eventually 'q' will hold as well. Because UPPAAL uses timed automata, this holds for delay transitions as well as action transitions.

Name	**Property**	**Equivalent to**
Possibly	`E<> p`	
Invariantly	`A[] p`	`not E<> not p`
Potentially always	`E[] p`	
Eventually	`A<> p`	`not E[] not p`
Leads to	`p --> q`	`A[] (p imply A<> q)`

In the above table the symbols have the following meanings:

E == ∃ == for a path there exists

A == ∀ == for all paths there exists

Not == ~ == ¬

and == ∧ == &&

or == ∨ == ||

implies == →

--> == leads to

iff == ↔

< > == eventually

[ ] == henceforth

The property ‘E<> p’ is satisfied for a timed transition system if and only if there is a sequence of transitions ‘s0-->s1 -->...-->sn’ where ‘s0’ is the initial state and ‘sn’ satisfies ‘p’.

The property ‘E[] p’ is satisfied if and only if there is a sequence of transitions ‘*s*0-->s1-->...-->si-->...’ for which p holds in all states ‘si’ for all ‘d: (Ln, vn + d)’ or where there is no outgoing transition from (Ln, vn).

The deadlock property may be satisfied for a state ‘(L, v)’ if and only if for all ‘d >= 0’ there is no transition successor to ‘(L, v + d)’.

A particular location may be evaluated to determine if the system exists in that location at a particular time step. Expressions of the form ‘P.l’, where ‘P’ is a process and ‘L’ is a location are satisfied in a state (L, v) if and only if ‘P.l’ is in ‘L’.

Temporal Analysis Results

The system was modified each time deadlock was encountered. Also, with the given trace, we can determine if the system was behaving as it was expected to and also we can determine if any implied scenario occurs because of the system modeling.

Advantage of UPPAAL Tool

During the modeling and verification of our model, we determined the following advantages of UPPAAL :

1. Inspite of its limited language, it covers almost entire timed automata.
2. It provides mechanism to use front end code like xml/java.
3. Excellent variable communication – synchronization is made possible just because of this.
4. Inspite of running as a Java applet, the program runs pretty fast.
5. Possible to do verification based changes. This feature helped a lot in our project.

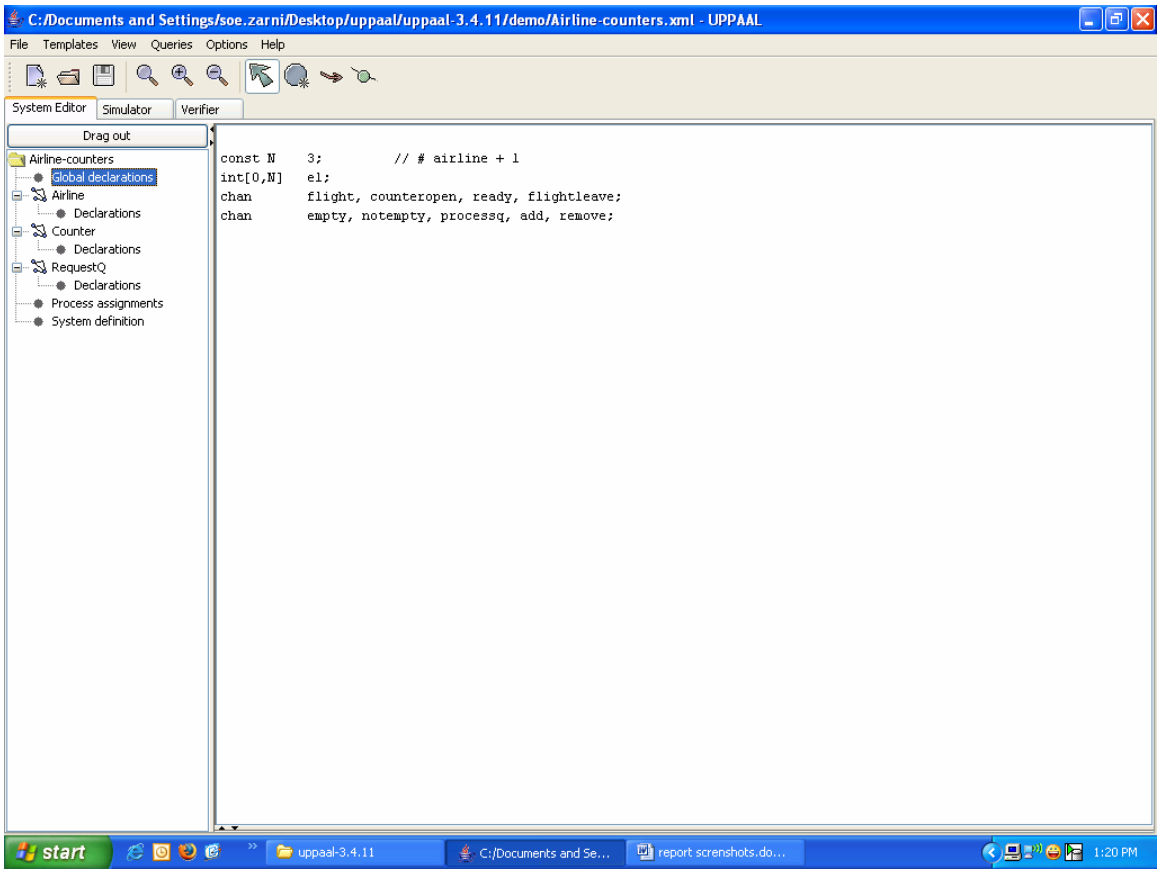
In Uppaal, systems are created, simulated and validated using three different stages;

1) System Editor – declarations, defining the systems, defining and assigning the process and building the systems and processes.
2) Simulator – auto-generated simulations for possible scenarios of the system built in System Editor.
3) Verifier – verification and validation of efficiency and effectiveness of the system using queries.

2 Airlines and 1 counter system; there are 2 airlines and the counter can be used as first come first serve basic.

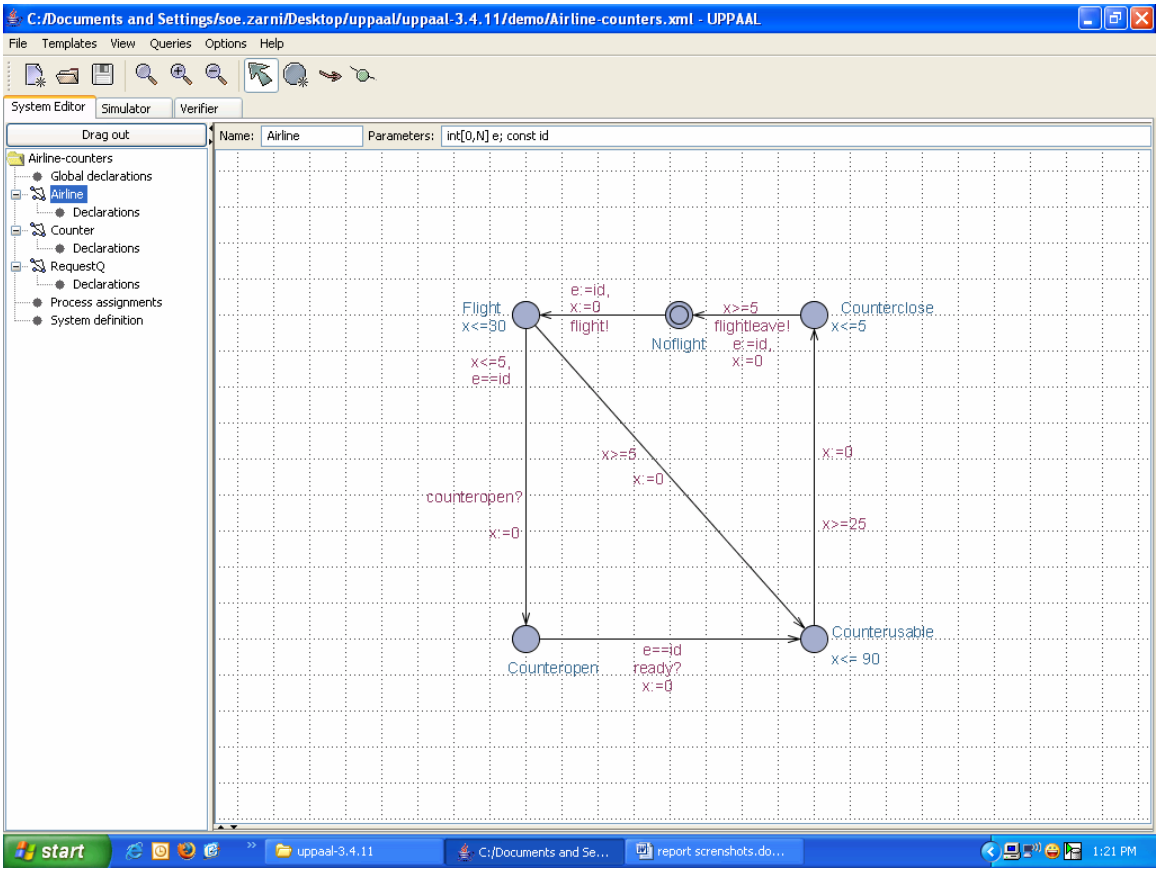
1) System Editor:

Declaration;



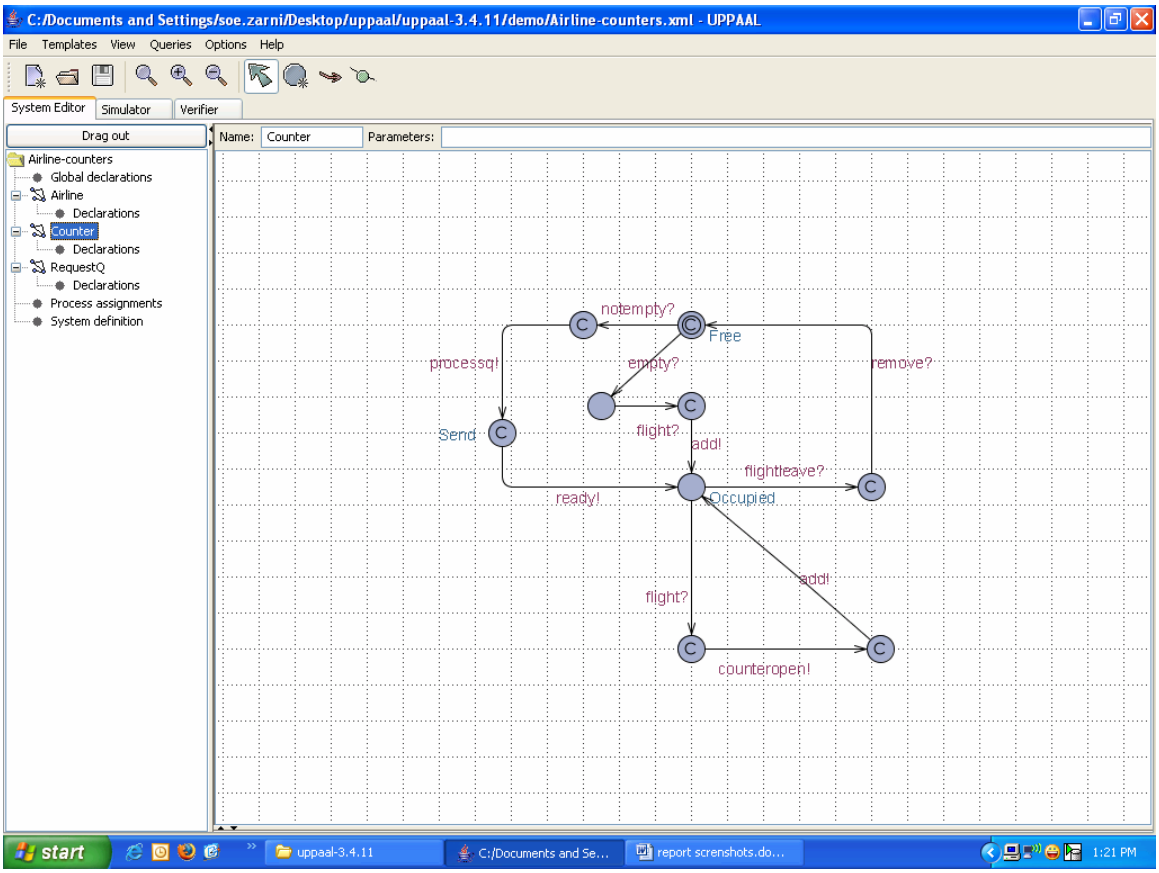
Defining the system flow and requirements for airlines;

- If an airline has a flight schedule and will make a request to use the counter 30 minutes in advance.
- If there is no reply of counter being used in 25 minutes, the airline can use the counter.
- If there is a reply saying another airline is using the counter, the requesting airline needs to wait until the counter is free.
- Any airline can use the counter for maximum of 90 minutes.
- There is maximum of 25 minutes to clear the counter.
- There is 5 minutes set up time for another airline to use the counter.



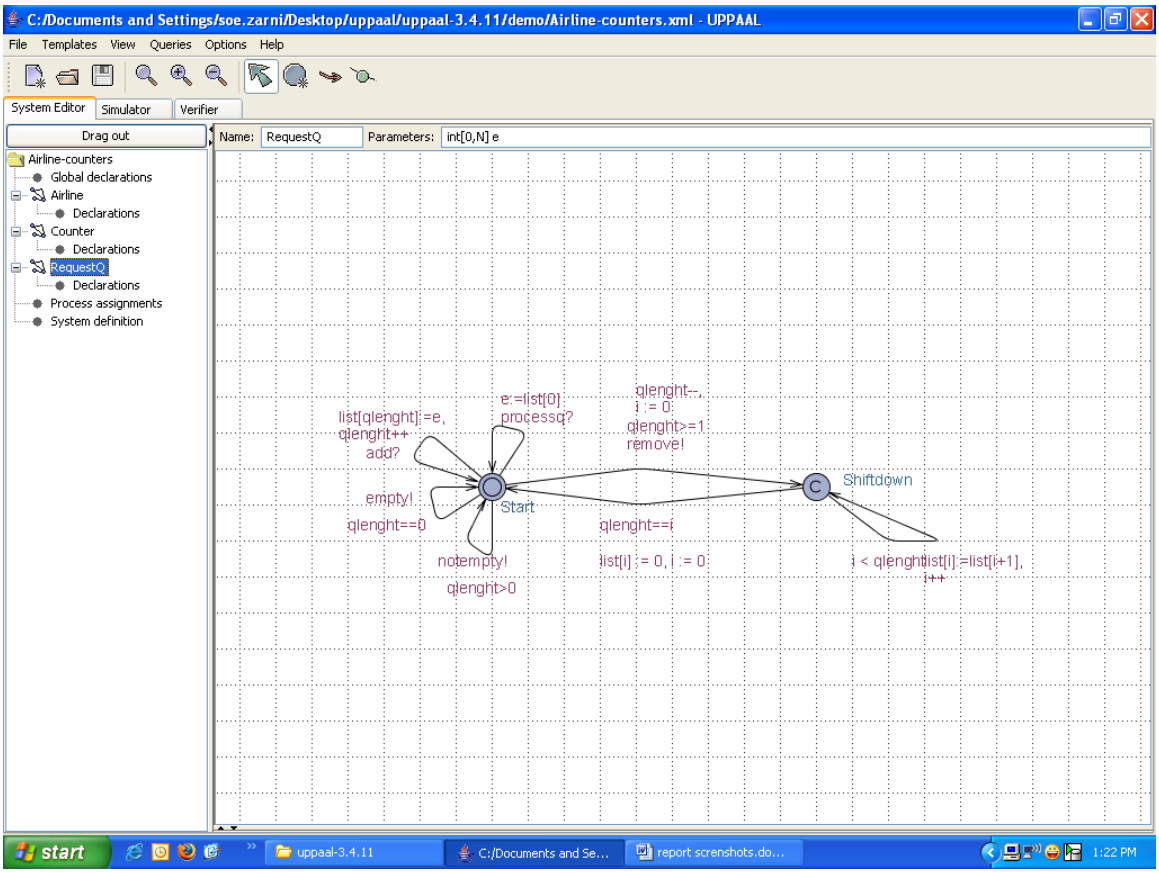
Defining the system flow and requirements for counter;

- If the counter is free, any airline can request and use the counter.
- If the counter is in use, requests from the airlines will be put into request queue.

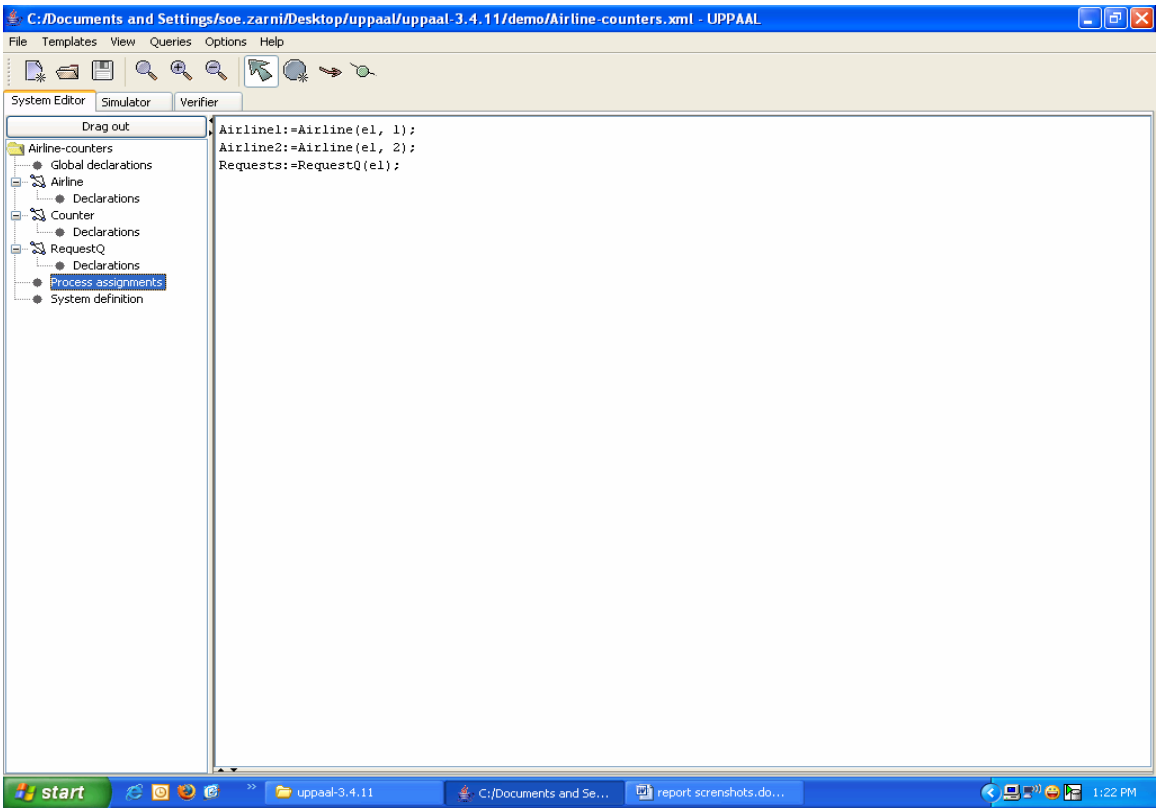


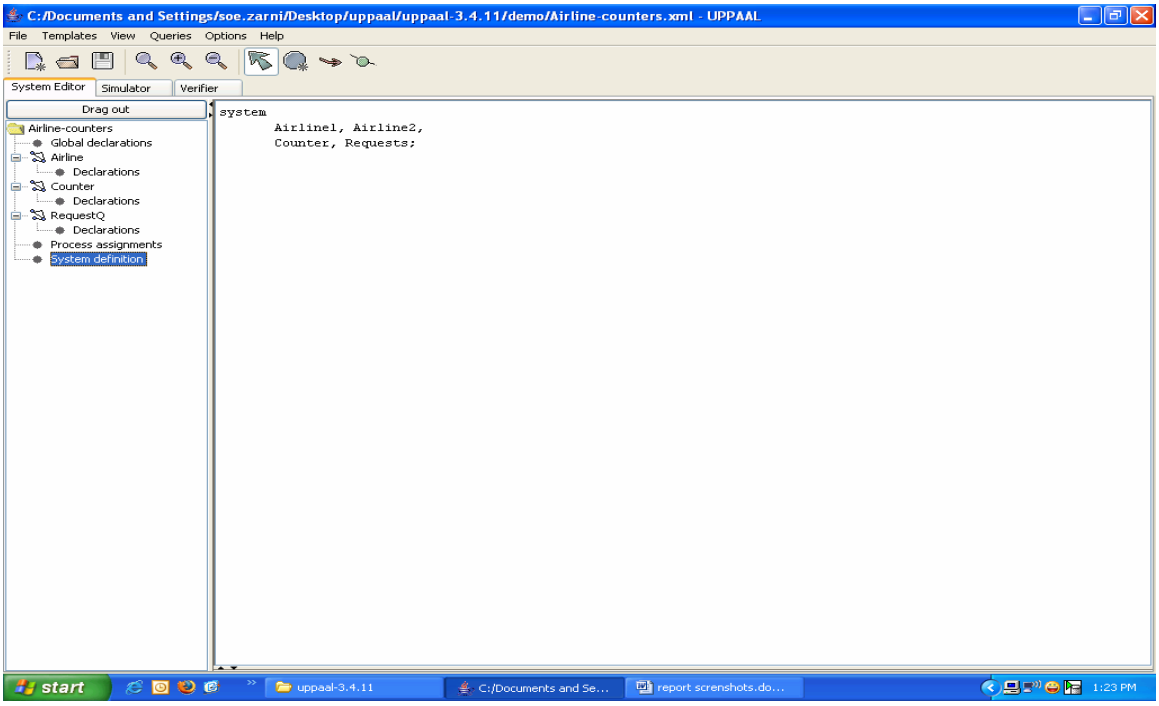
Defining the system flow and requirements for request queue;

- If there is(are) a request(s) and the counter is in use, the requests will be put in the queue and will be processed as first come first serve.



Process assignment and system definition;

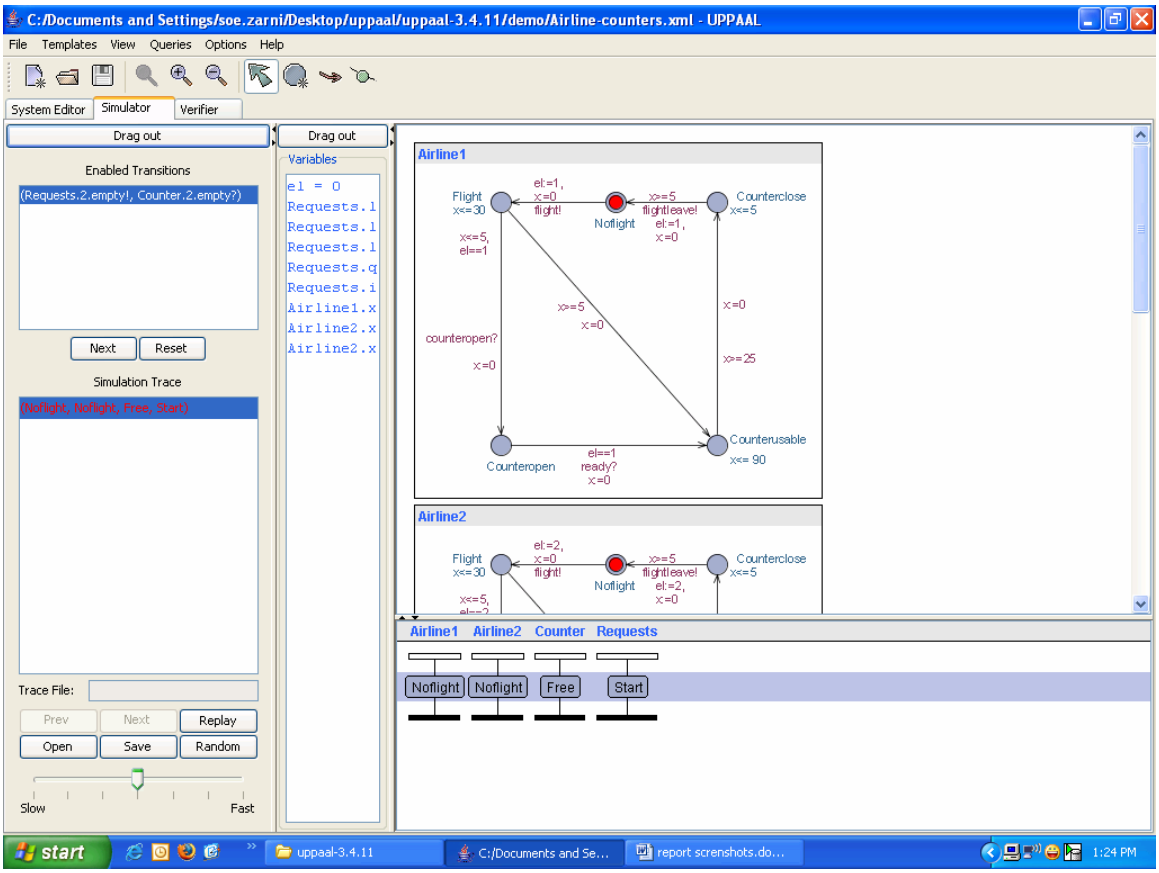


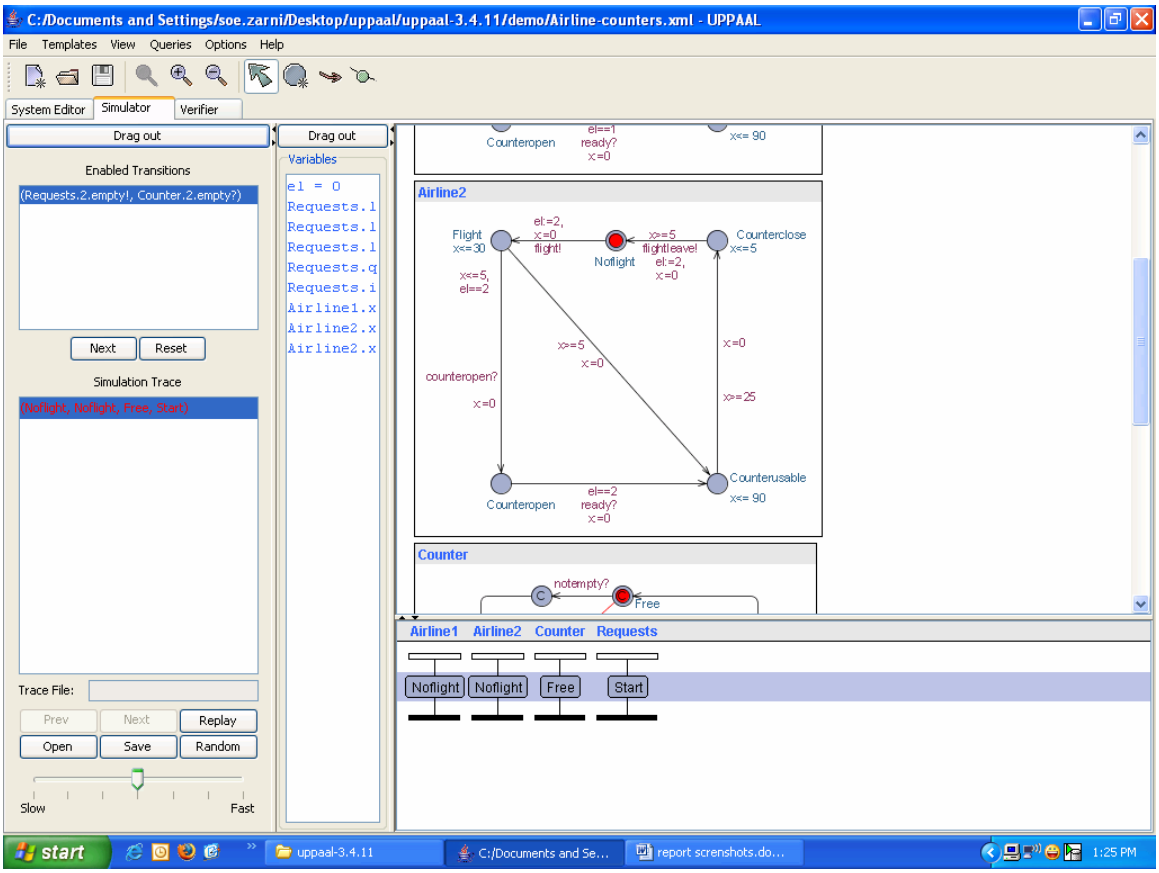


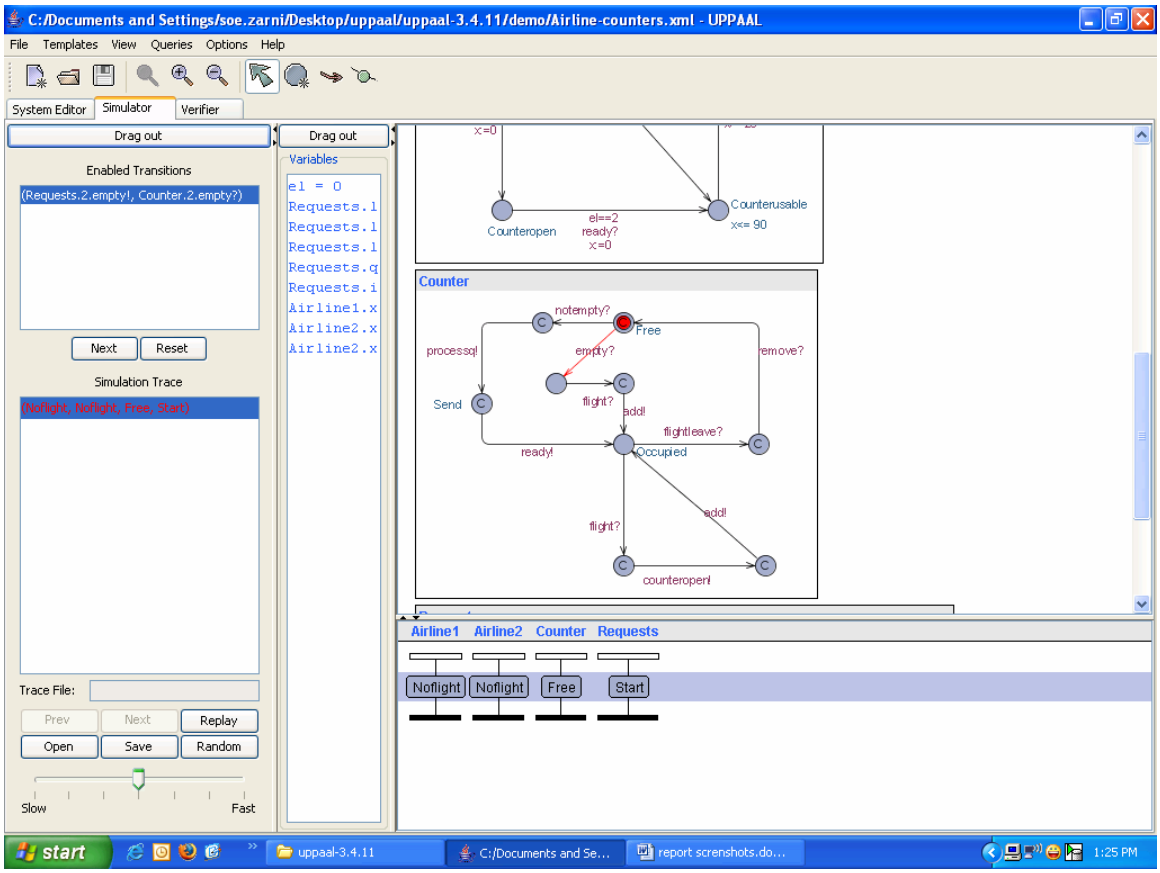
2) Simulator:

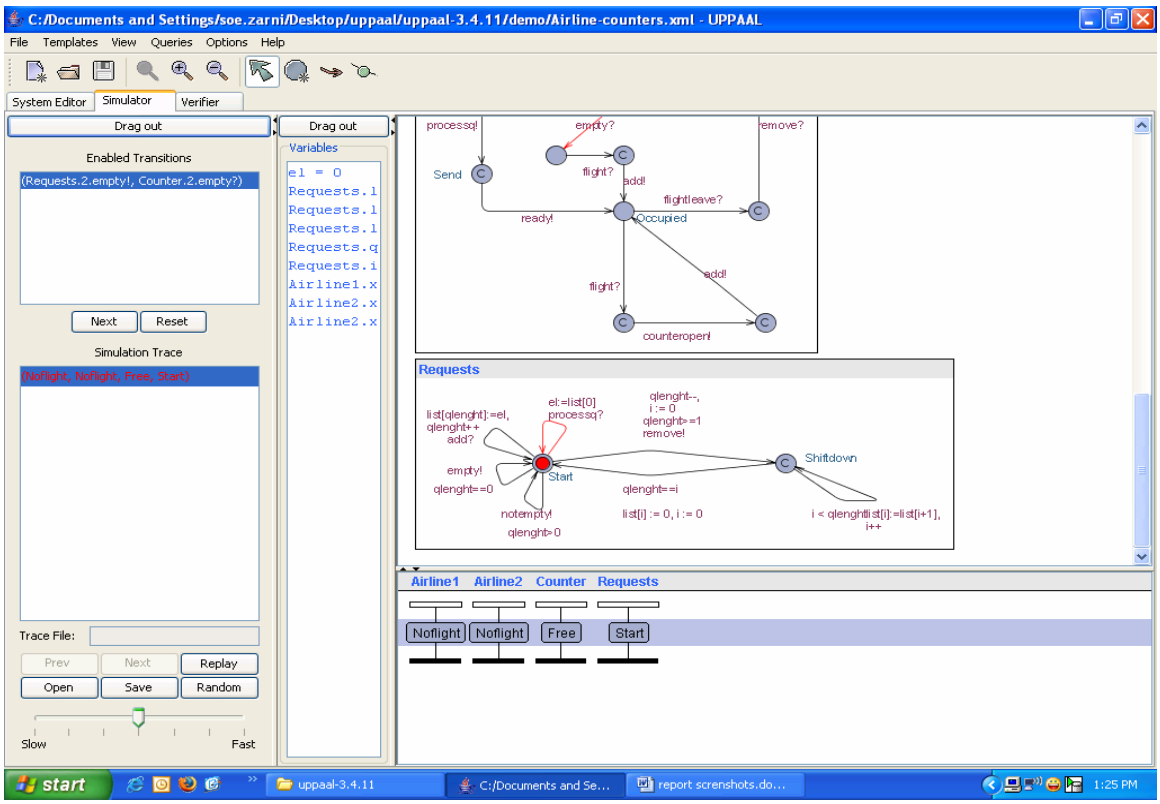
Examples;

All the systems are in idle states.

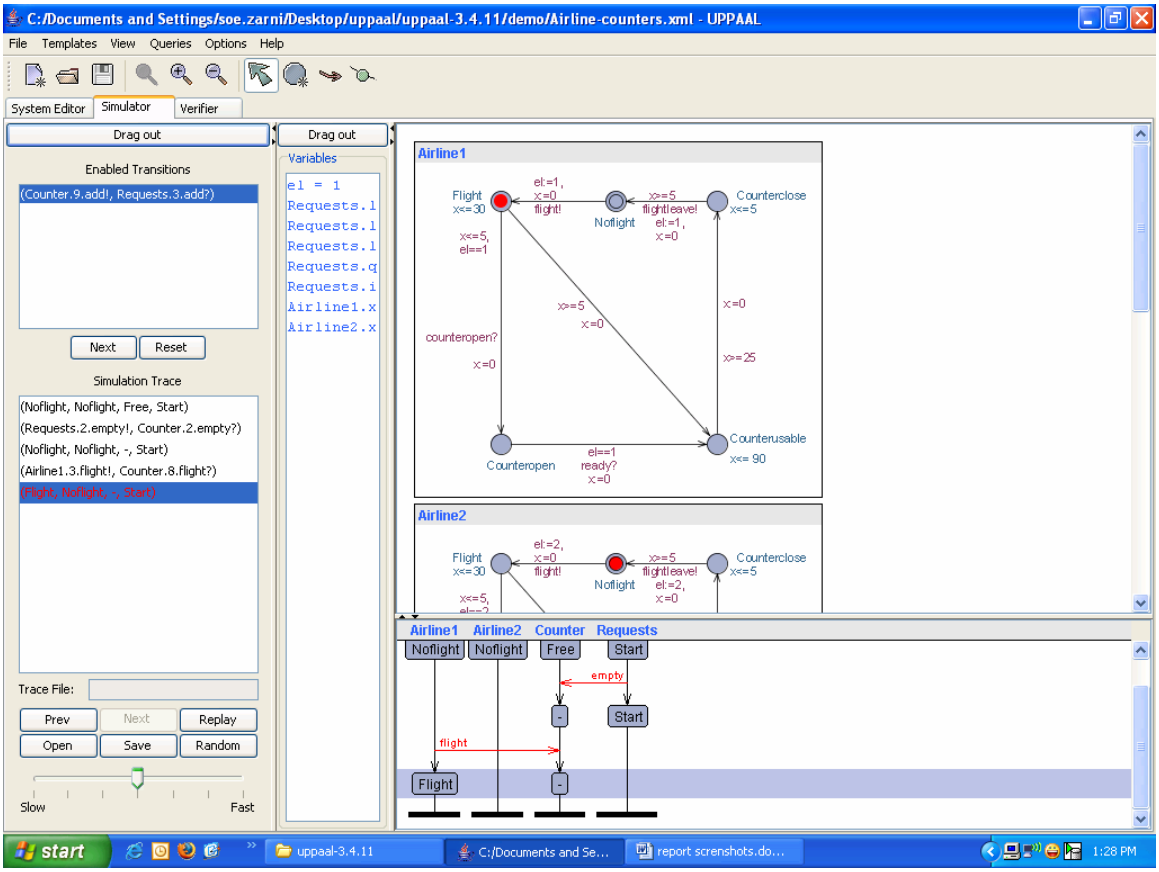



C:/Documents and Settings/soe.zarni/Desktop/uppaal/uppaal-3.4.11/demo/Airline-counters.xml - UPPAAL
File Templates View Queries Options Help
System Editor Simulator Verifier
Drag out
Enabled Transitions
(Requests.2.empty!, Counter.2.empty?)
Next Reset
Simulation Trace
(Noflight, Noflight, Free, Start)
Trace File:
Prev Next Replay
Open Save Random
Slow Fast
Drag out
Variables
el = 0
Requests.l
Requests.l
Requests.l
Requests.q
Requests.i
Airline1.x
Airline2.x
Airline2.x
Counteropen
el==1
ready?
x=0
x<= 90
Airline2
Flight
x<=30
el:=2,
x:=0
flight!
Noflight
x>=5
flightleave!
el:=2,
x:=0
Counterclose
x<=5
x<=5,
el==2
x>=5
x:=0
x:=0
counteropen?
x:=0
x>=25
Counterusable
x<= 90
Counteropen
el==2
ready?
x=0
Counter
notempty?
Free
Airline1 Airline2 Counter Requests
Noflight Noflight Free Start
start
uppaal-3.4.11
C:/Documents and Se...
report screnshots.do...
1:25 PM

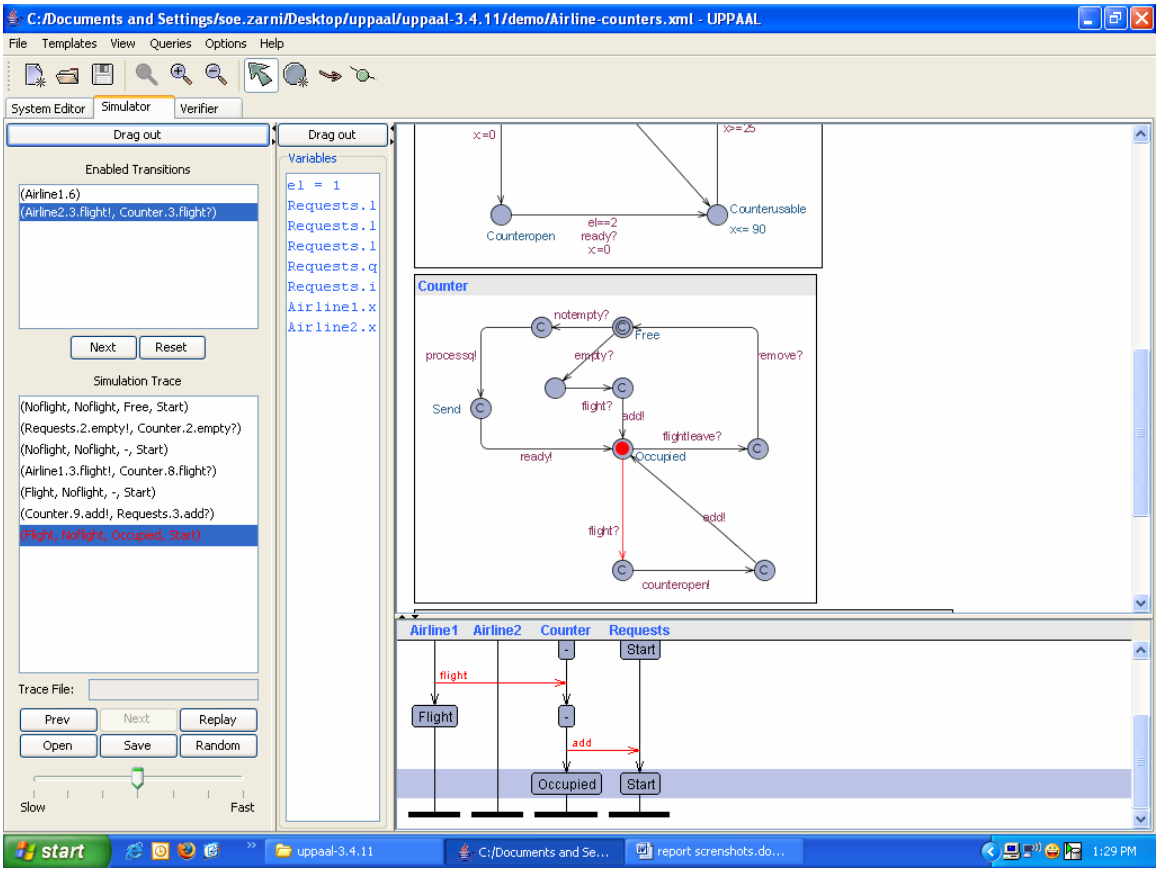

C:/Documents and Settings/soe.zarni/Desktop/uppaal/uppaal-3.4.11/demo/Airline-counters.xml - UPPAAL
File Templates View Queries Options Help
System Editor Simulator Verifier
Drag out
Enabled Transitions
(Requests.2.empty!, Counter.2.empty?)
Next Reset
Simulation Trace
(Noflight, Noflight, Free, Start)
Trace File:
Prev Next Replay
Open Save Random
Slow Fast
Drag out
Variables
el = 0
Requests.l
Requests.l
Requests.l
Requests.q
Requests.i
Airline1.x
Airline2.x
Airline2.x
x=0
Counteropen
el==2
ready?
x=0
Counterusable
x<= 90
Counter
notempty?
Free
processq!
empty?
remove?
Send
flight?
add!
flightleave?
ready!
Occupied
add!
flight?
counteropen!
Airline1 Airline2 Counter Requests
Noflight Noflight Free Start
start
uppaal-3.4.11
C:/Documents and Se...
report screnshots.do...
1:25 PM


C:/Documents and Settings/soe.zarni/Desktop/uppaal/uppaal-3.4.11/demo/Airline-counters.xml - UPPAAL
File Templates View Queries Options Help
System Editor Simulator Verifier
Drag out
Enabled Transitions
(Requests.2.empty!, Counter.2.empty?)
Next Reset
Simulation Trace
(Noflight, Noflight, Free, Start)
Trace File:
Prev Next Replay
Open Save Random
Slow Fast
Drag out
Variables
el = 0
Requests.l
Requests.l
Requests.l
Requests.q
Requests.i
Airline1.x
Airline2.x
Airline2.x
processq!
empty?
remove?
Send
flight?
add!
flightleave?
ready!
Occupied
add!
flight?
counteropen!
Requests
list[qlenght]:=el,
qlenght++
add?
el:=list[0]
processq?
qlenght--,
i := 0
qlenght>=1
remove!
Shiftdown
empty!
qlenght==0
Start
qlenght==i
notempty!
qlenght>0
list[i] := 0, i := 0
i < qlenghtlist[i]:=list[i+1],
i++
Airline1 Airline2 Counter Requests
Noflight Noflight Free Start
start
uppaal-3.4.11
C:/Documents and Se...
report screnshots.do...
1:25 PM

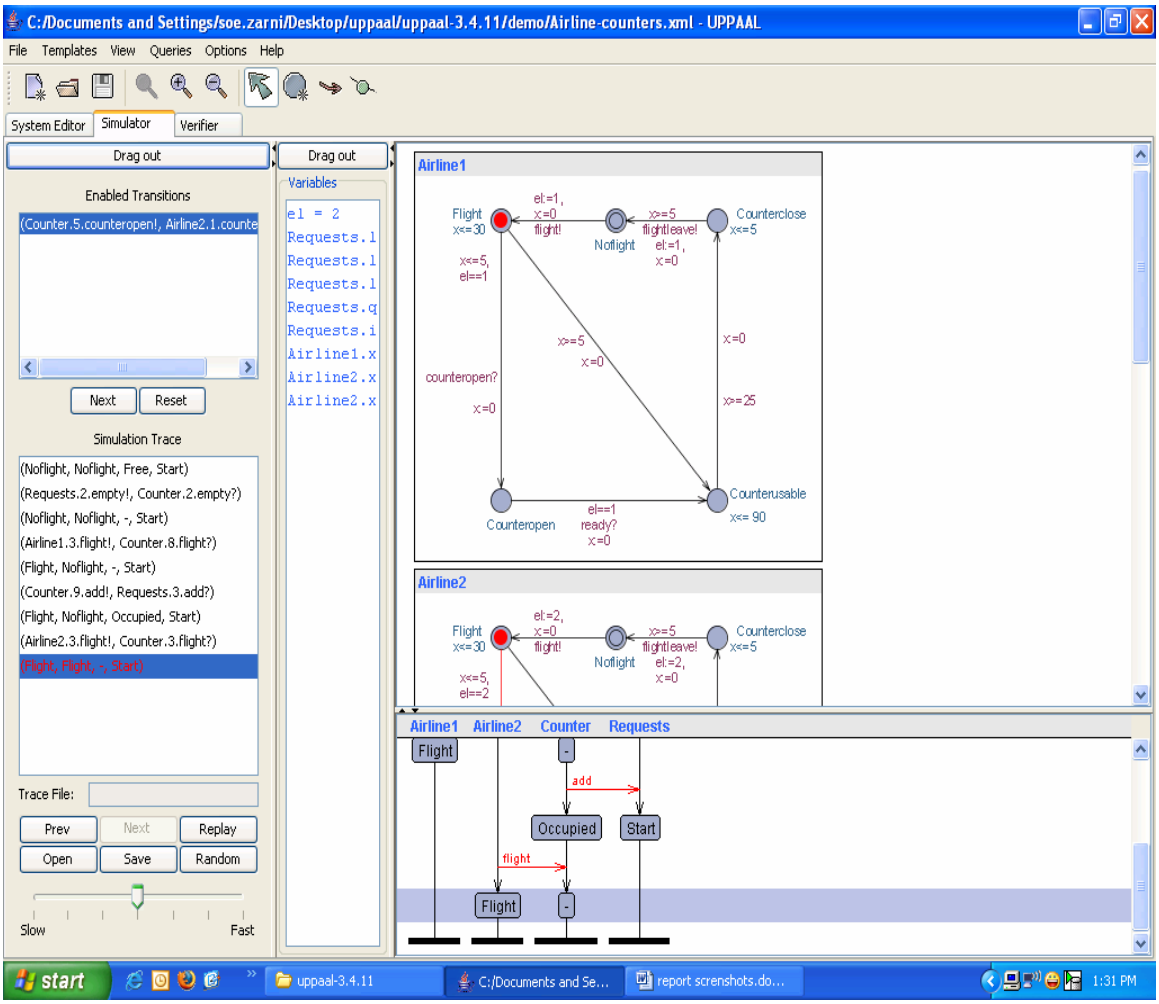
a) There's a flight and request to use the counter from Airline 1.



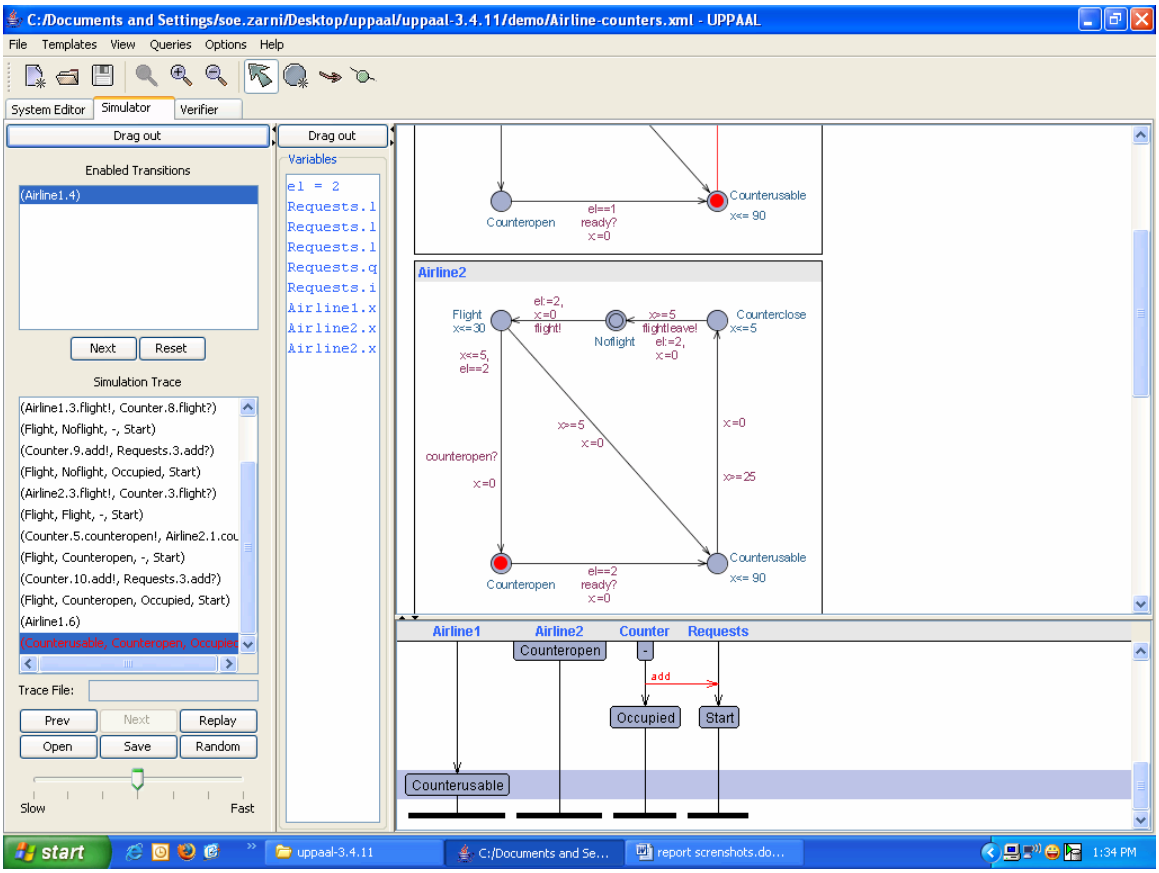
Now airline 1 occupied the counter as no one was using it.



b) Airline 1 makes the request and then Airline 2 also makes a request.



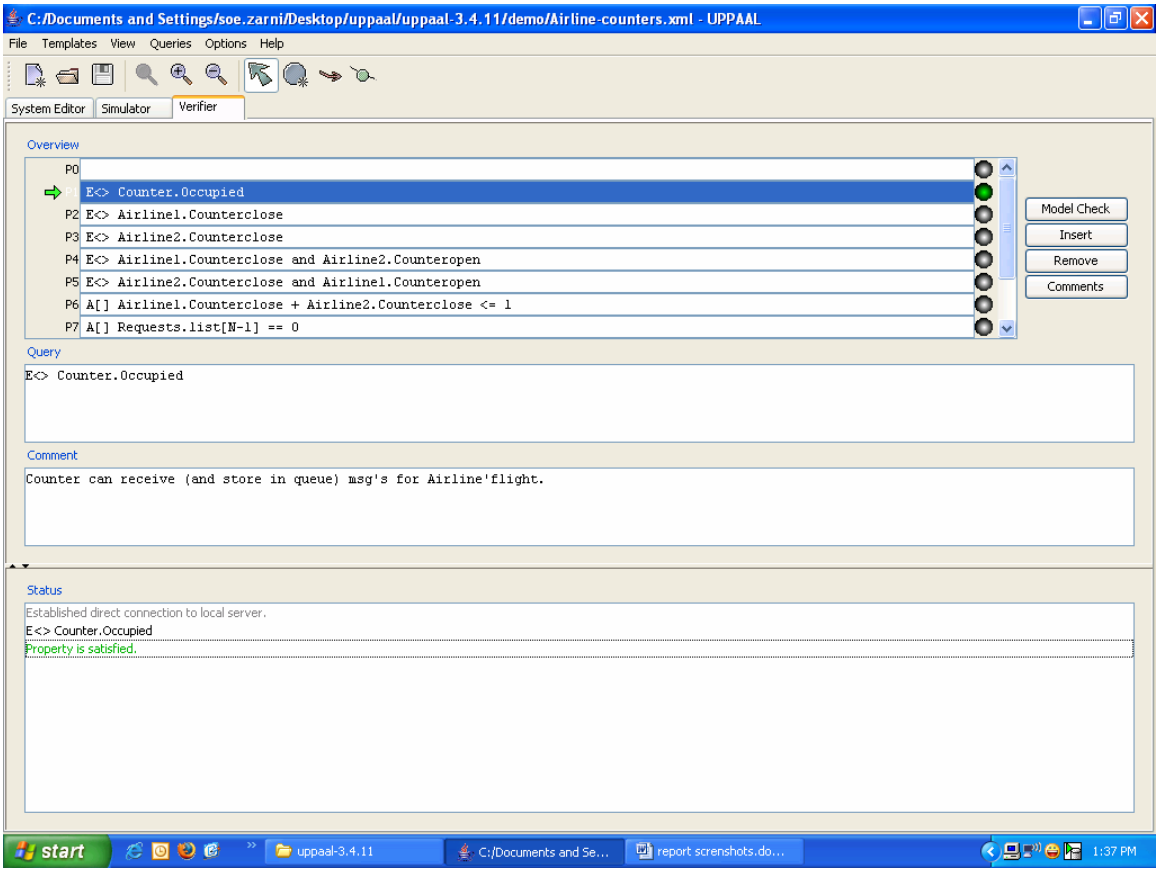
As Airline 1 made the request first, airline 1 is using the counter and airline 2 has to wait. Counter add the request from airline 2 to request queue.



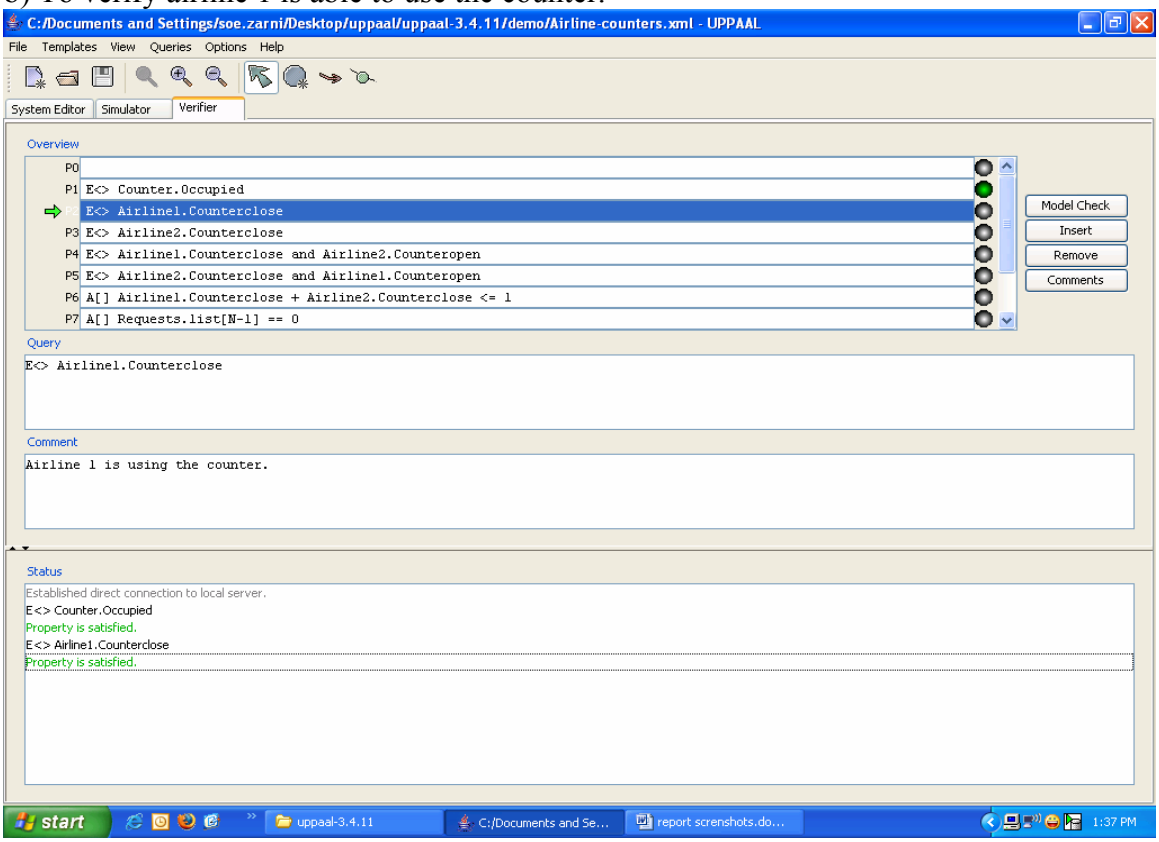
3) Verifier:

Various queries were made to verify and validate the system.

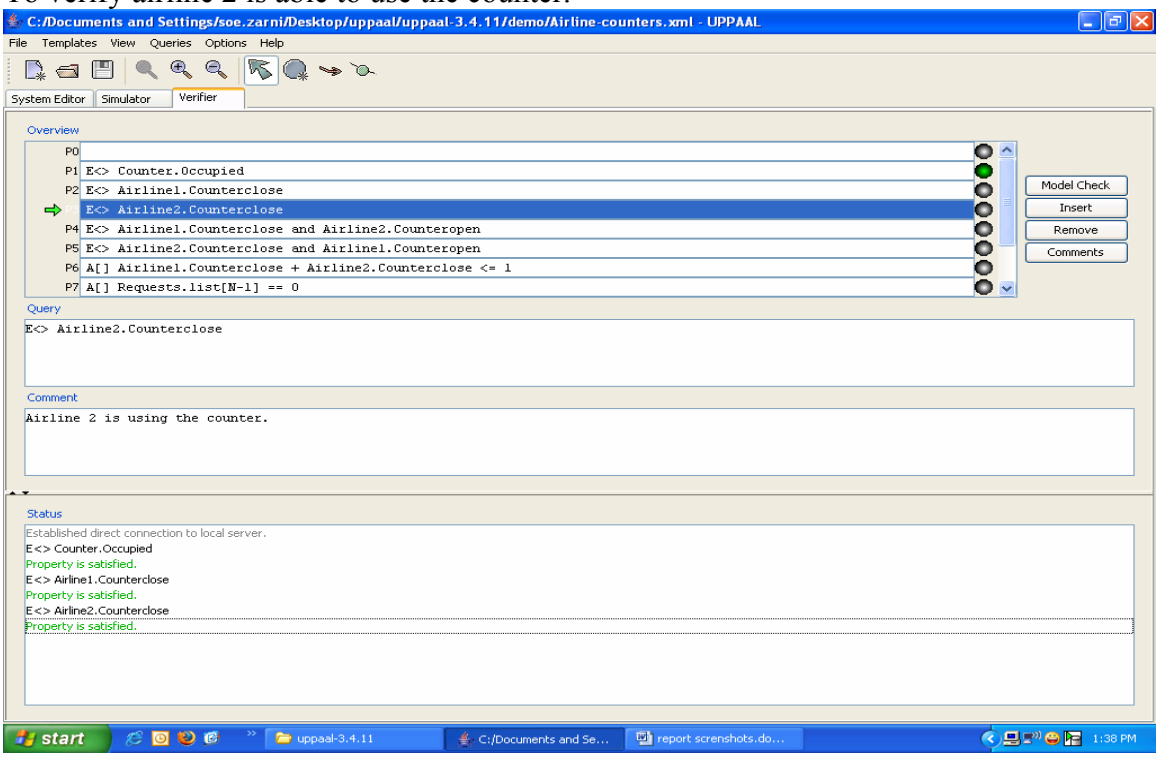
a) This query is used to verify the counter can be occupied or not.
Status – the property is satisfied.



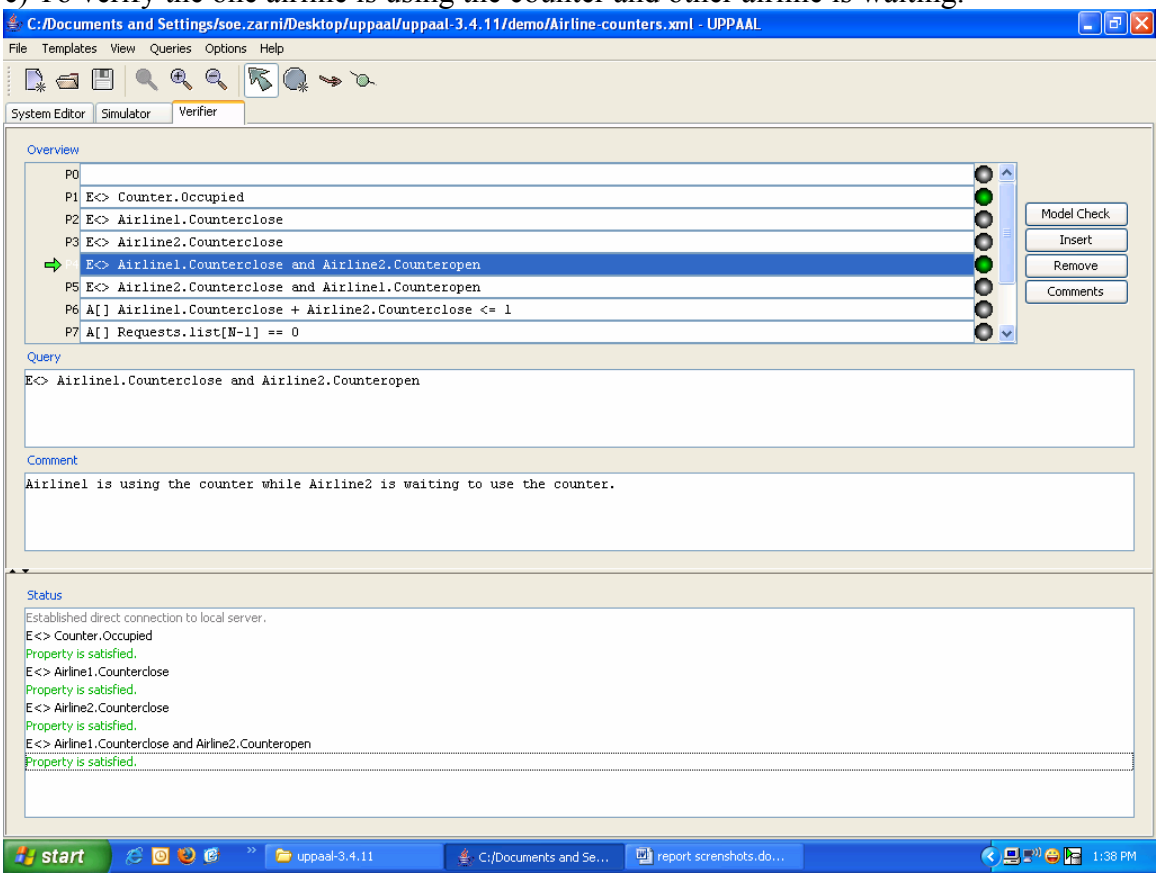
b) To verify airline 1 is able to use the counter.



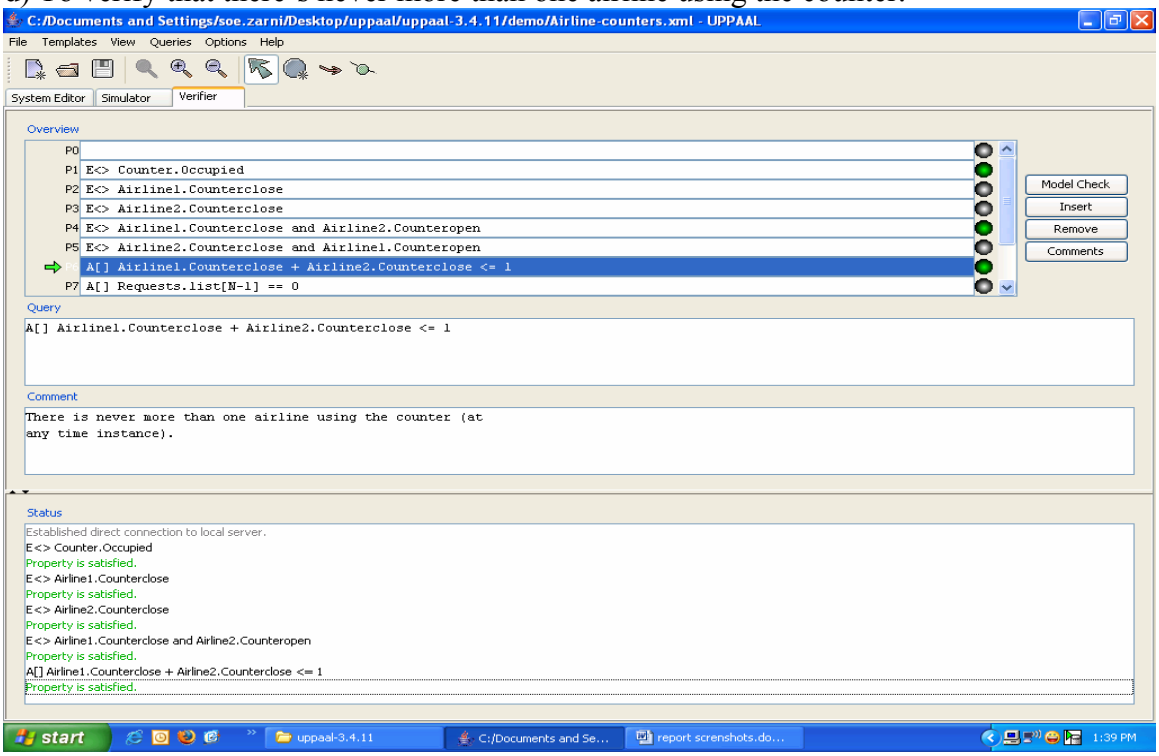
To verify airline 2 is able to use the counter.



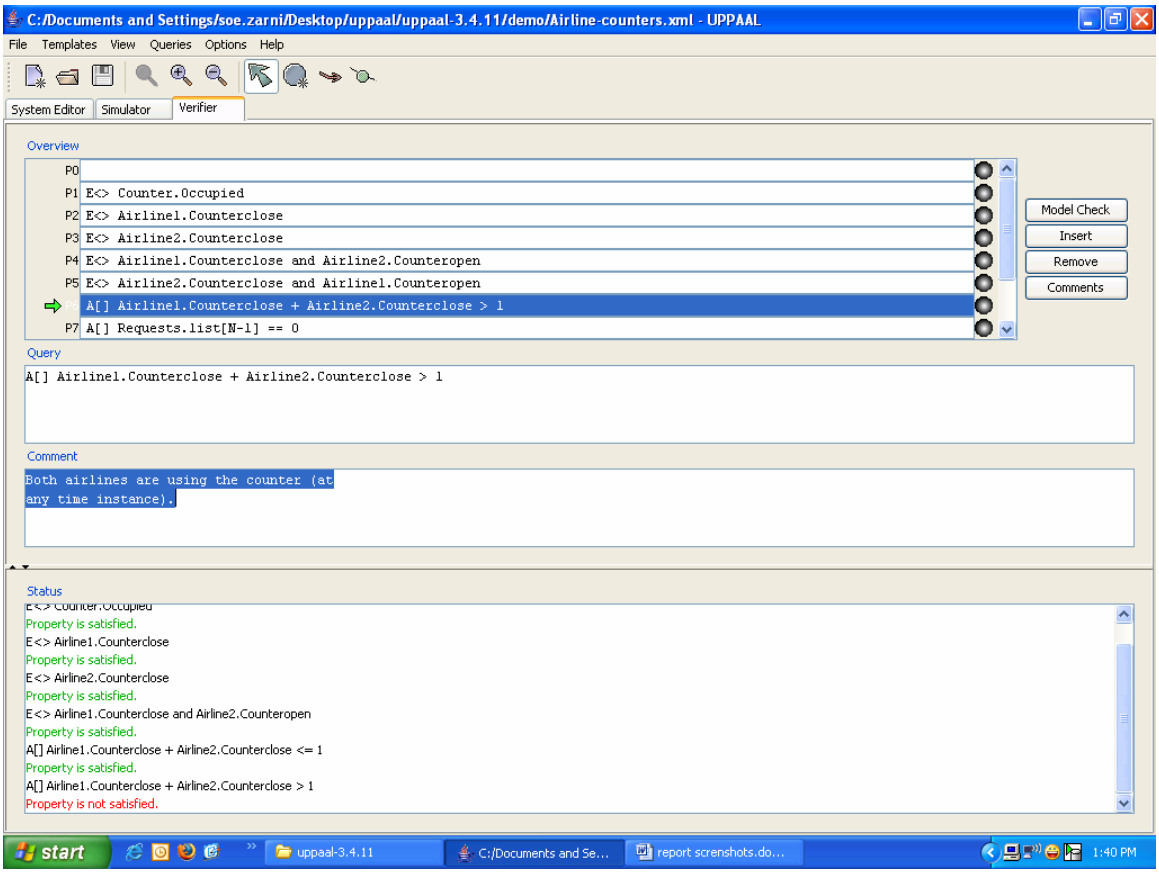
c) To verify the one airline is using the counter and other airline is waiting.



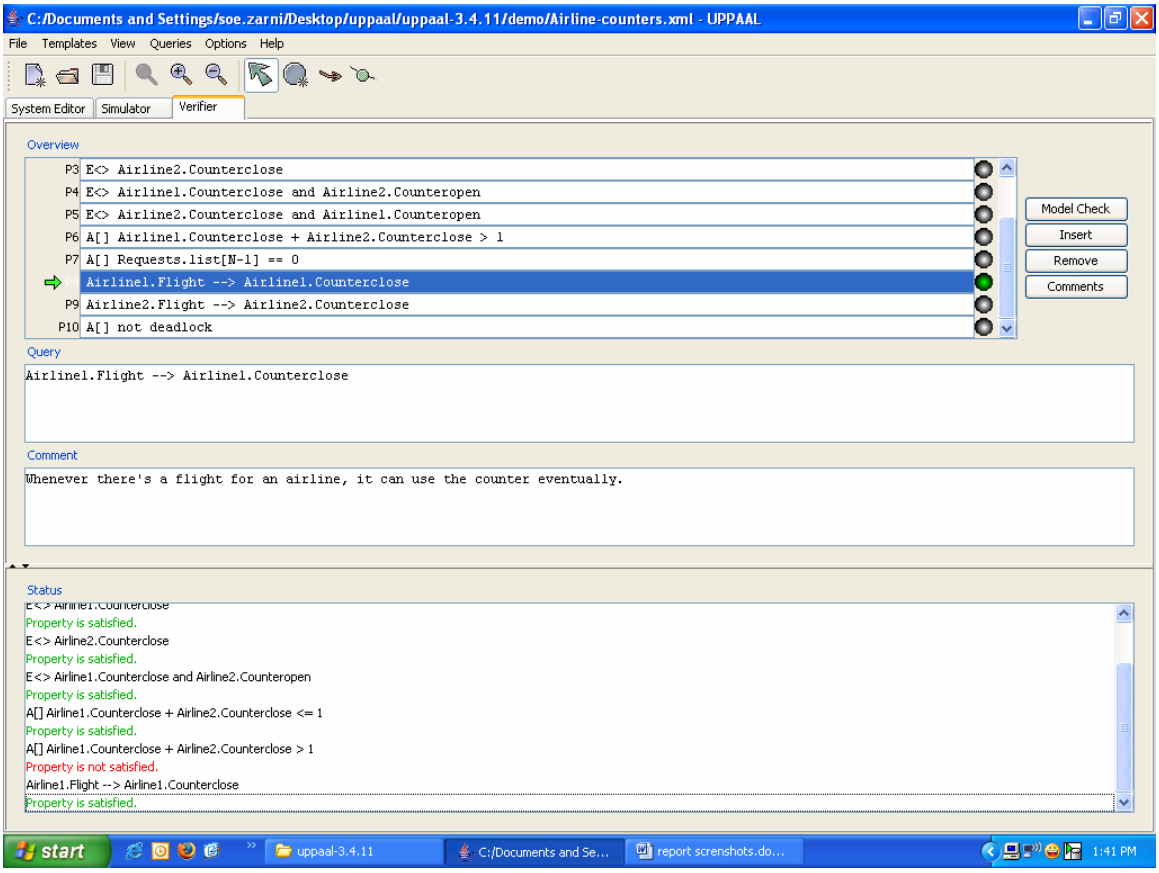
d) To verify that there's never more than one airline using the counter.



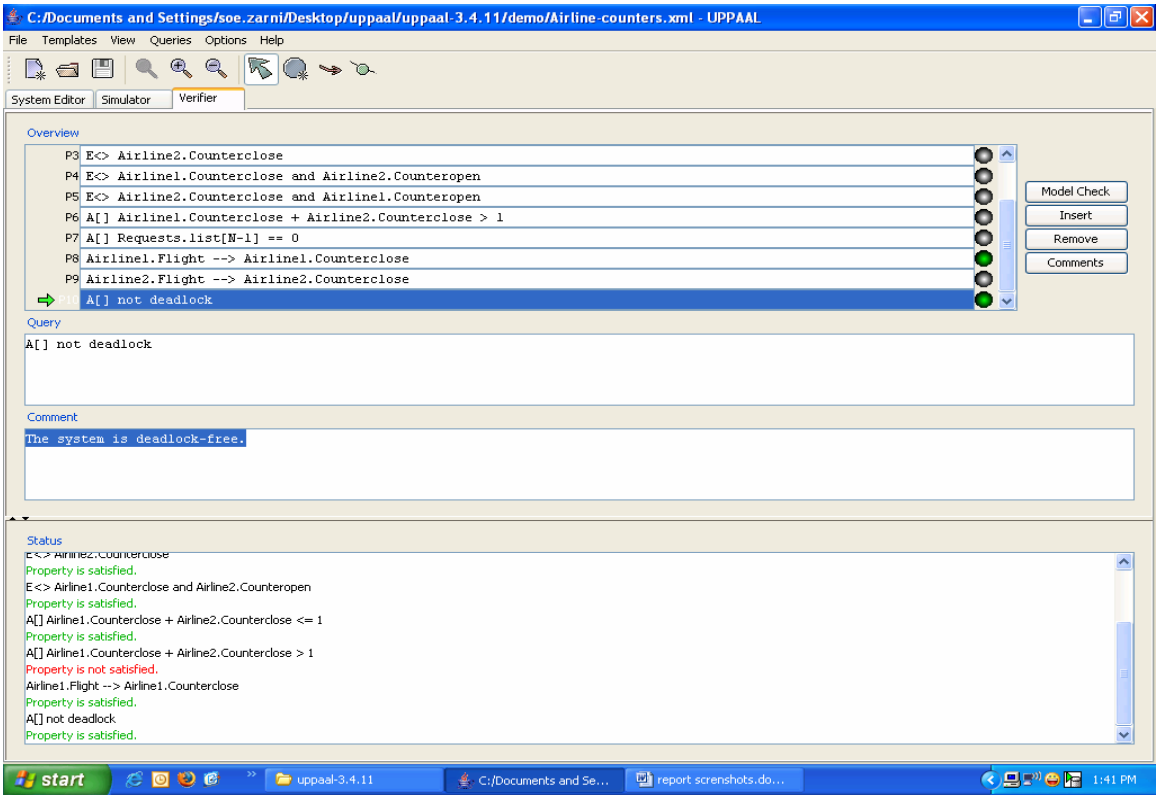
e) This is to verify whether TWO airlines can use the counter at the same time interval or not. This should not happen as we defined in requirements that only one airline can use the counter at a time.
Status shows that property is NOT satisfied.



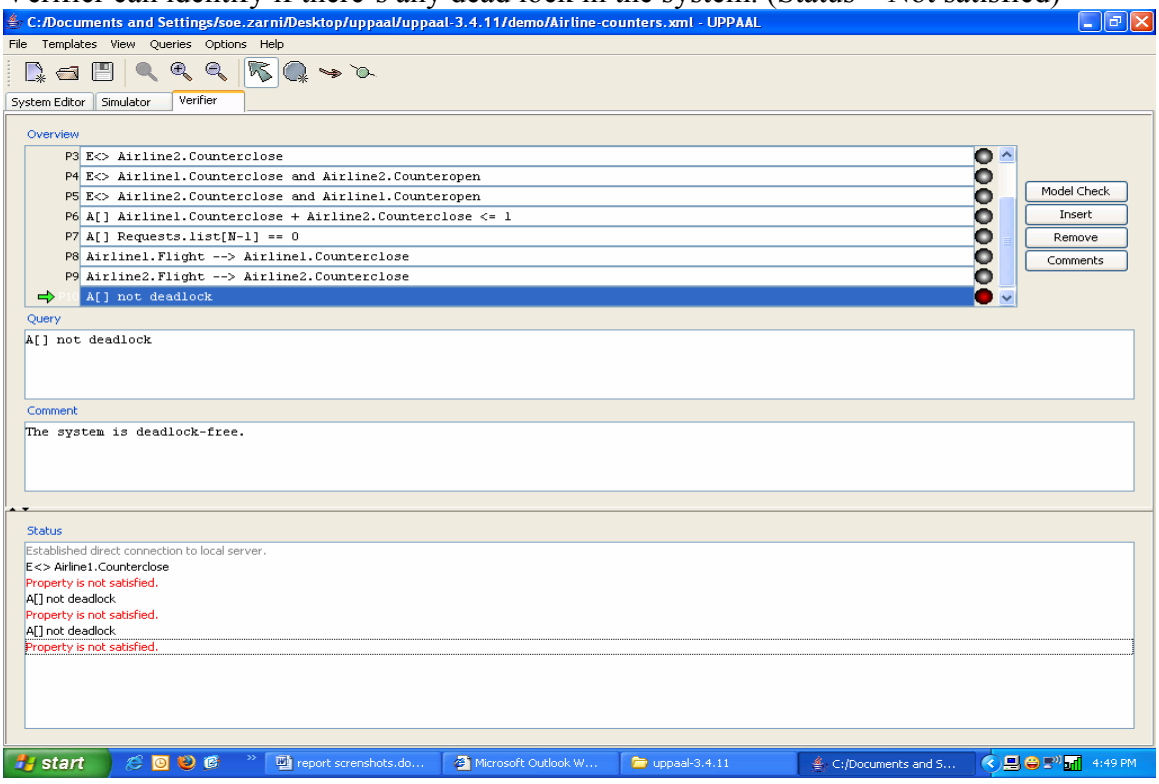
f) This is to verify that whenever there's a request from an airline, the request has been processed and the airline can use the counter eventually.



g) This verifies that there's NO dead lock in the system.

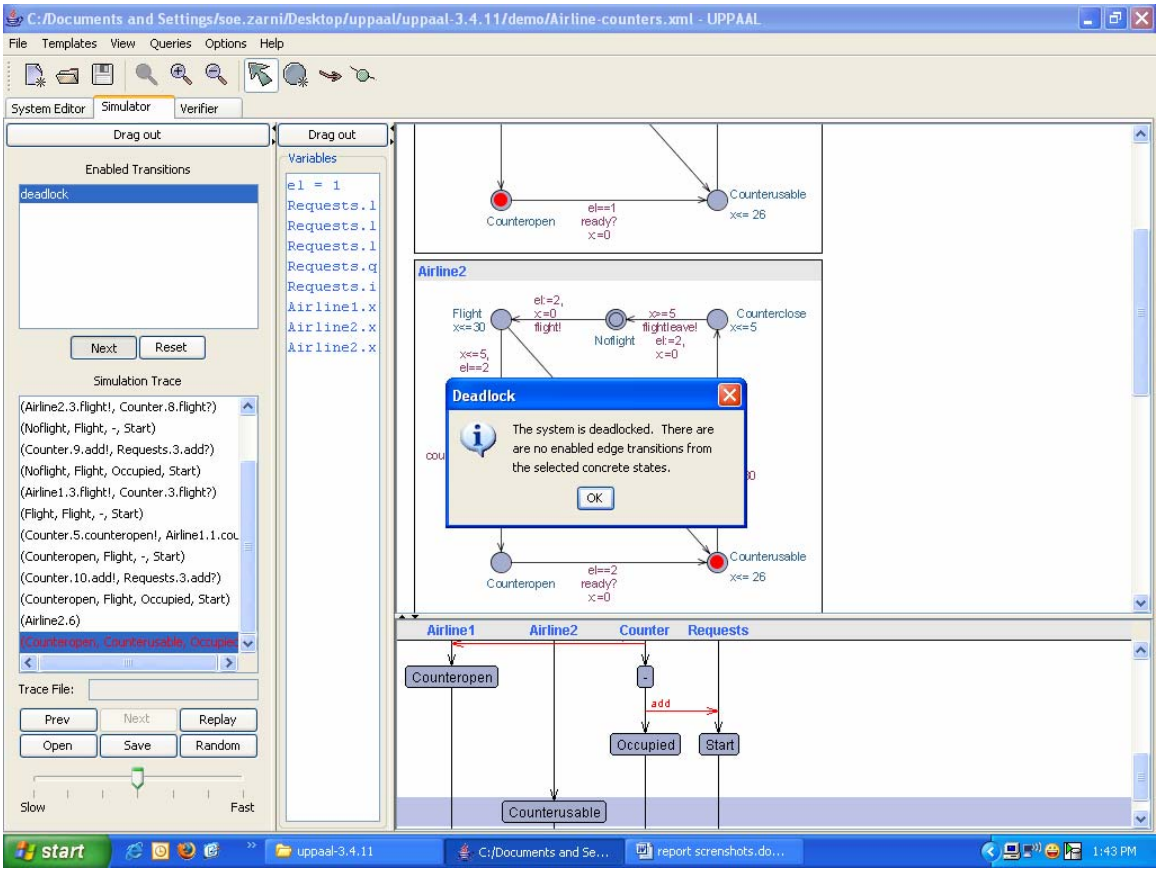


Verifier can identify if there's any dead lock in the system. (Status – Not satisfied)

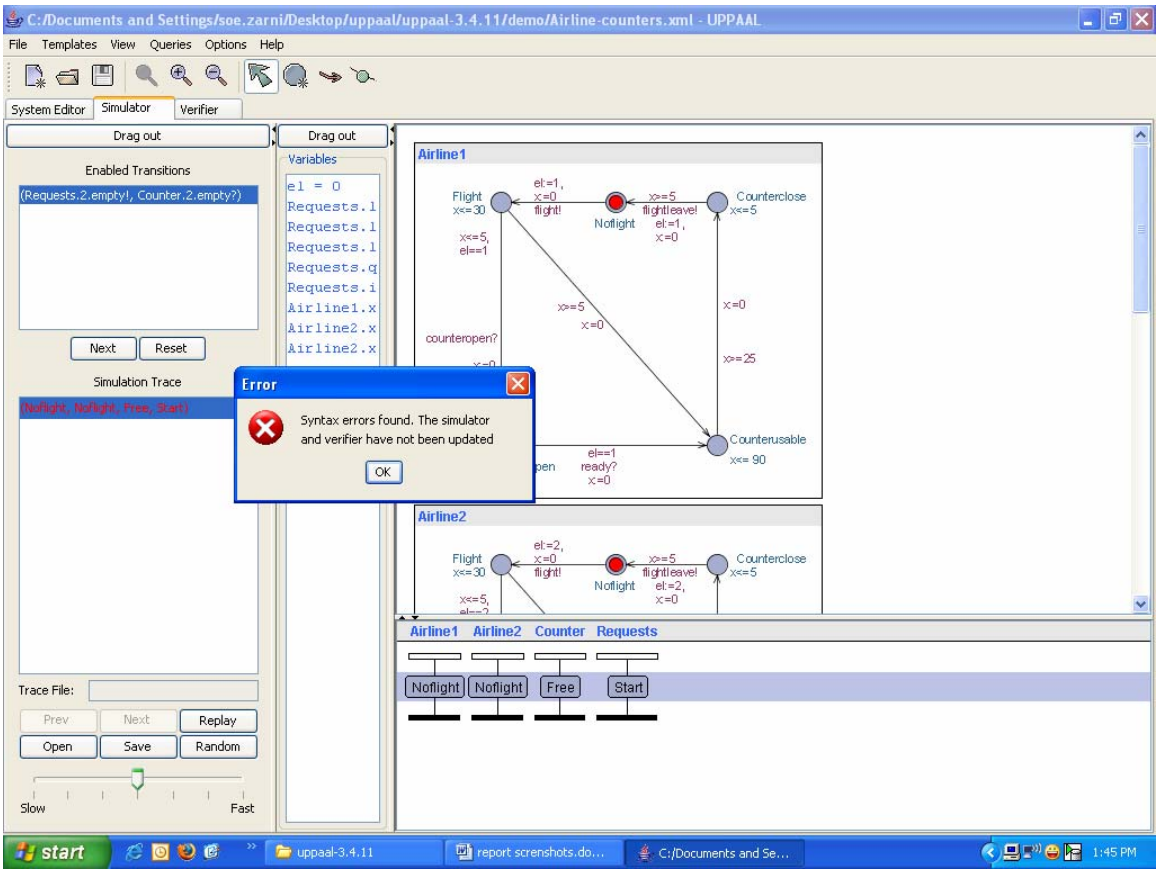


Errors traceability:

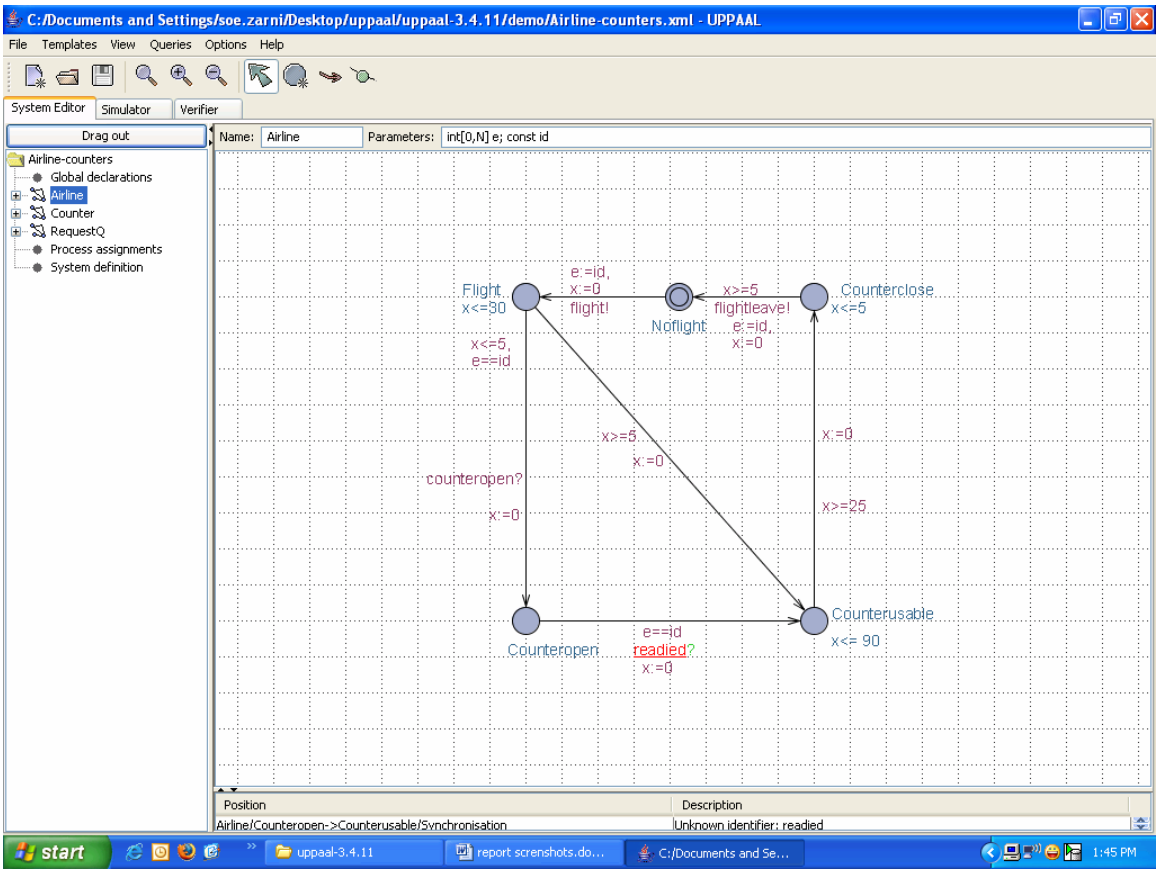
Simulator can point out if there's any deadlock.



Simulator can also point out if there's any error in the system.



The syntax error in system editor is shown in red and underlined.



Conclusion and Future Work

Viewpoints from the perspective of passengers and airlines have been developed. With this individual view points, the concurrent system is modeled. The issues of passenger's queue length, number of counters to be used and effective use of the limited space by two airlines by inter-dependent communication have been studied.

The challenging features of the project are

1) Concurrent behavior

2) Multiple airline multiple passenger destinations

3) Dynamic system structure

The system analysis, verification/validation of the concurrent system has been carried out. The tool UPPAAL has been used to validate them along with timing constraints. Together with the spatial and temporal constraints, we've carried out a study of a specific issue of the airline industry which can further be enhanced to include all actual timing and optimization constraints. UPPAAL CORA can be used to specify cost functions to each of the automata and a timing based cost model can be generated.

Hence, we have designed a system, carried out spatial and temporal analysis of the system and validated/verified the spatial and temporal logic. Any changes suggested by inconsistencies in the logic were fixed.

References

Mark Austin: ENSE 621 Systems Modeling and Analysis and Lecture Notes

Learning UML, O'Reilly 2003

Magee and Kramer, Concurrency: State Models and Java

UML Diagrams
http://pigseye.kennesaw.edu/~dbraun/csis4650/A&D/UML_tutorial/diagrams.htm

UML Tutorial in 7-days http://odl-skopje.etf.ukim.edu.mk/uml-help/

Mark Austin: ENSE 621 Systems Modeling and Analysis and Lecture Notes

Magee and Kramer, Concurrency: State Models and Java

Mark Austin, Design Structure Matrices Tutorial

P. BELLINI, R. MATTOLINI, and P. NESI, Temporal Logics for Real-Time System Specification

Johan Bengtsson and Fredrick Larsson UPPSALA University, A Tool for Automatic Verification of Real-Time System UPPALL.

Gerd Behrmann Kim Guldstrand LarsenAalborg University, DENMARK, Verification of Non-Functional Properties.

Kim G. Larsen, Paul Petterson and Wang Yi - University of Bermingham, The model-checker UPPAAL

Kim G. Larsen , Paul Pettersson , and Wang Yi - University of Bermingham, UPPAAL in a Nutshell

M. Oliver M¨ oller – BRICS PhD School, Structure and Hierarchy in Real-Time Systems

Luca Tesei, Specification and Verification using Timed Automata.

Natalia Sidorova, Modelling Timed Systems

Mamta Kothavale, Verifying a Burglar Alarm System Using UPPAAL

Roger T. Alexander Mark R. Blackburn, Ph.D., Specification-Based Analysis and Testing

Simon Burton, Automated Generation of High Integrity Test Suites from Graphical Specifications

Roberto Sebastiani, Introduction to Formal Methods for SW and HW Development

UPPALL Aalborg University, Denmark, http://www.uppaal.com/

http://ctp.di.fct.unl.pt/SLMC/manual.pdf

Abrash, M., BSP Trees, Dr. Dobbs Sourcebook, 20(14), 49-52, may/jun 1995.

Dadoun, N., Kirkpatrick, D., and Walsh, J., The Geometry of Beam Tracing, Proceedings of the ACM Symposium on Computational Geometry, 55--61, jun 1985.

Chin, N., and Feiner, S., Near Real-Time Shadow Generation Using BSP Trees, Computer Graphics (SIGGRAPH '89 Proceedings), 23(3), 99--106, jul 1989.

Chin, N., and Feiner, S., Fast object-precision shadow generation for area light sources using BSP trees, Computer Graphics (1992 Symposium on Interactive 3D Graphics), 25(2), 21--30, mar 1992.

Chrysanthou, Y., and Slater, M., Computing dynamic changes to BSP trees, Computer Graphics Forum (EUROGRAPHICS '92 Proceedings), 11(3), 321--332, sep 1992.