Flight Scheduling System

Table of Contents

Chapter 1

Introduction

1.1 Background

Worldwide, countries have been reporting a strong demand for air travel. A recent report released by the Federal Aviation Administration, predicts air travel to double in the next 20 years. Singapore Changi Airport reported a 9.4% year-on-year increase in passenger traffic in August 2013. Air traffic movements included a total of 29,600 landings and take-offs, an increase of 8.2% compared to the past year.

These strong statistics suggests a rapid growth in air travel in the near future. To cater to the increasing trend in air travel, Changi Airport must maximise its resources and efficiency. It was announced in the Budget 2013 that Changi Airport will have its 3rd runway ready by the end of the decade to increase flight handling capacity and cope with the growing number of flights. The new runway will be able to double the passenger handling capacity of Changi Airport. This calls for a more capable and robust flight scheduling system that to keep the 3 runways optimized.

There are several aspects of the flight scheduling system. In this project, the focus was placed on investigating the impact of the increase in runway on flight handling capacity and scheduling the takeoff and landing of flights on the runways.

1.2 Problem and Motivation

This project aims to modify a simplified version of a flight scheduling landing and takeoff system. The motivation behind this project is to explore the relationship between flights served and the new runway addition so that the best schedule for the flights can be identified. The runways are the main bottleneck and resource constraint. It is not economical to increase the number of runways in a land scarce island of Singapore.

In addition, constraints such as fuel limitations will be added to simulate an actual flight scheduler. Other resource constraints include gates (dependent on the number built in each airport terminal), air tugs which are vehicles that help to reverse a plane from the gate and the time limitation for all planes to be handled.

Among the many states generated by Process Analysis Toolkit (PAT), the best scenario (goal) will indicate the concurrent and sequential processes that will best utilize the resources. Besides finding the maximum number of planes that can be served (take off and land) within a set amount of time without any plane running out of fuel, other interesting values can be derived from the model. This will be further discussed in the next chapter.

The difficulty of the system lies in ensuring that the scheduled flight trace does not have any accidents. For example, the model must ensure that are only one plane on each runway at any time. If there are two planes are on the same runway, this would lead to plane collisions (Tenerife plane crash, 1977).


SINGAPORE AIRLINES

Chapter 2

System Modelling

Three CSP models are used to address the primary, secondary and tertiary goals of this project; With one additional model that models to achieve the tertiary goal in a different approach.

Primary Goal: investigate if flights can be doubled with the expansion of 2 to 3 runways Secondary Goal: schedule flights with general constraints of fuel, gates and air tugs Tertiary Goal: scheduling flights with priority on different fuel level and emergency situations

2.1 Primary Model

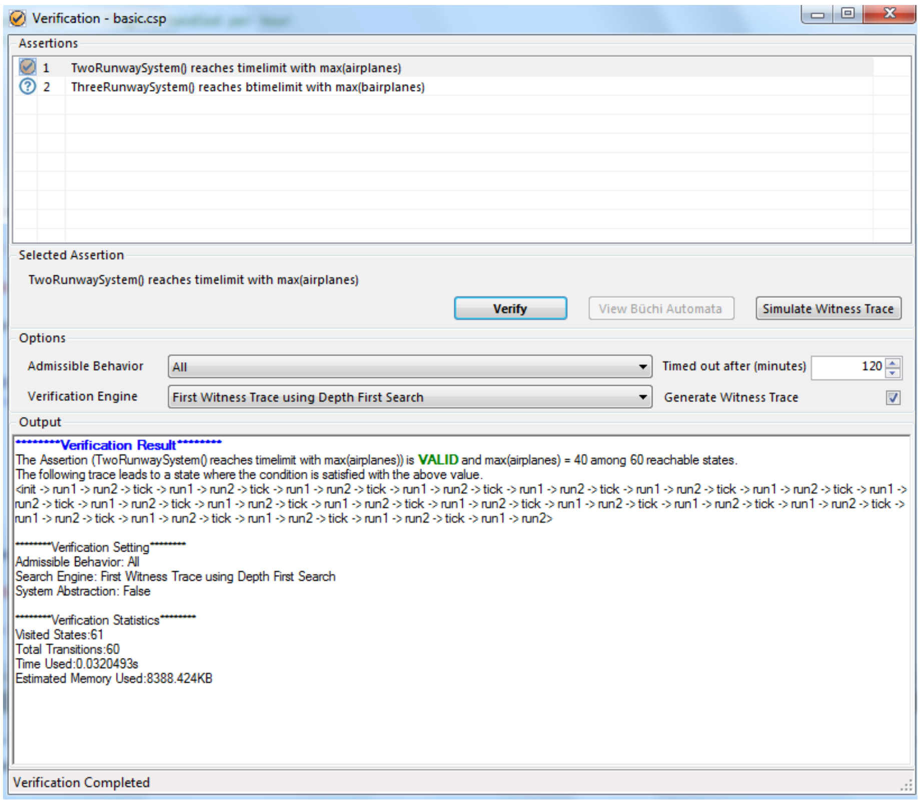
2.1.1 Model Description

This primary model presents a CSP Module with basic runways to investigate if flights can be doubled with the expansion of 2 to 3 runways. This model maintains two variables: runways and time. The airplanes are assumed to be infinitely requesting for use of runway. Each use of runway for takeoff or landing will take up 3 minutes. The time is incremented by 3 at every stage. The assertion finds the maximum number of planes that can be served within an hour for 2 runways and for 3 runways.

```
#define timelimit time<60; //set time limit of 60 seconds
#assert TwoRunwaySystem reaches timelimit with max(airplanes);
#assert ThreeRunwaySystem reaches timelimit with max(airplanes);
```

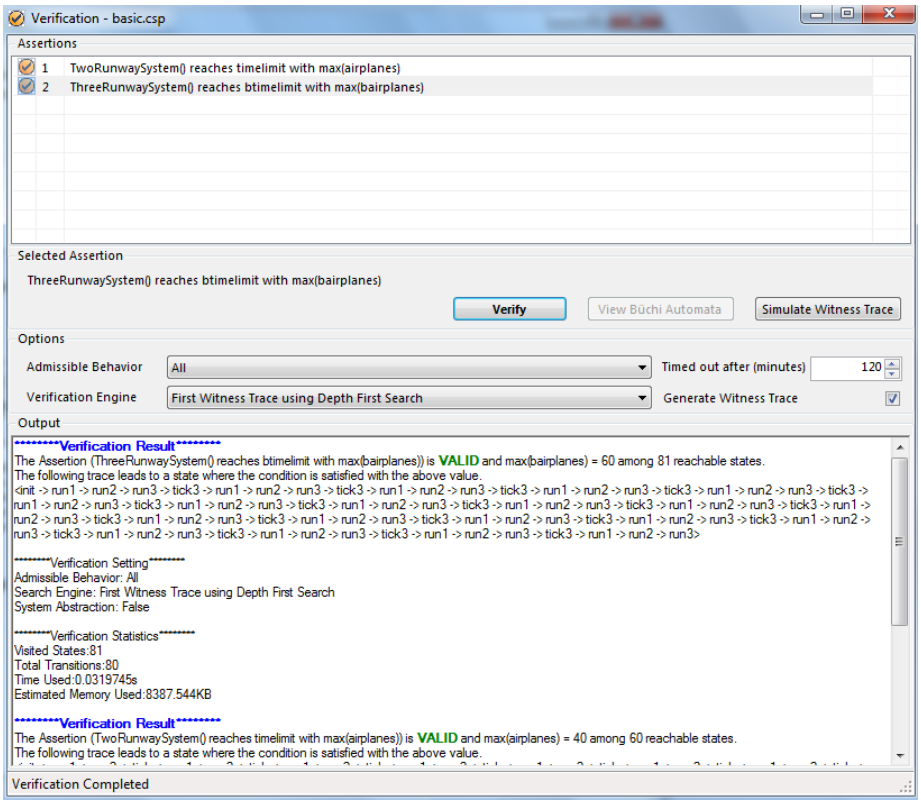
2.1.2 Experiments and Discussion

Running PAT verifies that with 2 existing runways, Changi Airport can handle up to 40 flights/hour which is very similar to the actual statistics of 37 flights/hour.



The main goal of this model is to verify if 3 runways can handle doubled flight capacity of 80 planes/hour. And the results show that the maximum flight handling capacity becomes 60 planes/hour which is not doubled. Perhaps Changi Airports passenger handling

capacity can be doubled with larger planes (such as the Airbus 380) that can potentially carry more passengers per flight on top of building a new runway.



2.2 Secondary Model

2.2.1 Model Description

In this model, resource constraints are introduced. The rules that this improved model includes fuel, parking gates, runway and airtug constraints.

- Fuel Constraint

An aircraft landing cannot run out of fuel. The plane must have at least 85 gallons to land, for it takes 80 gallons to touchdown and 5 gallons to taxi to the gate. The value 85 gallons is based on the Boeing 747 plane.

- Gate Constraint

Before an aircraft can land, there must be an available gate for it. It is undesirable for an aircraft to land on the runway but do not have an available gate to park at.

- Runway Constraint

Only 1 aircraft can use 1 runway at a time. If the runway is in use, no other landing or takeoff can occur on that runway. An aircraft must use a runway to land or takeoff.

- Airtug Constraint

An aircraft must have 1 airtug before it can taxi to takeoff.

```
22  //Process variables
23  var fuel = plane_fuel; // All flying airplanes assumed to have plane_fuel of 300
24  var toland[num_flying] = [plane_flying(num_flying)]; // All flying airplanes set to status 1
25  var runways = num_runways;
26  var free_gates = num_gates-num_parked; // gates available
27  var gates = num_gates;
28  var airtugs = num_tugs;
29
30  //Assertion variables
31  var timer=0;
32  var crash=0;
33  var planelanded=0;
34  var planetakeoffed=0;
```

fuel: In this model, the fuel is initialised to an arbitrary defined amount of fuel. This variable keeps track of the least amount of fuel among all planes.

toland[]: This array maintains the state of the planes flying waiting for touchdown. The possible states are: plane flying, on runway and parked.

runways, freegates, airtugs: This variable keeps count of the number of runways that are free and likewise for freegates and airtugs.

timer: This is the variable that maintains the amount of time that has passed.

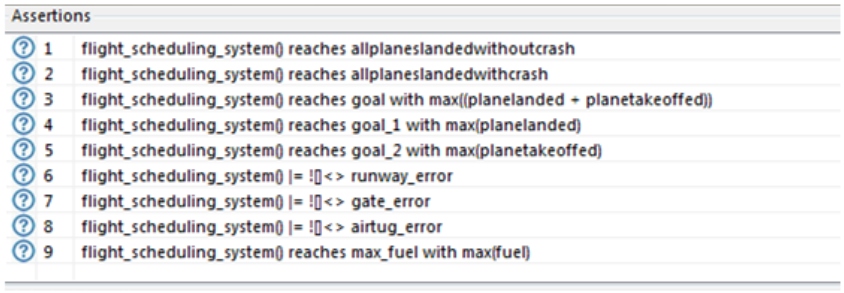
crash: This variable counts the times where a plane has run out of sufficient fuel to land.

planelanded, planetakeoffed: These variables maintain the count of planes which have successfully landed or taken off.

timer, crash, planelanded and planetakeoffed are assertion variables that can determine

the success of the scheduling algorithm that is taken by PAT.

Assertions



Assertion 1 is to find a scheduling sequence where all planes will land without any running out of fuel (crashing). If it is not valid, we look at Assertion 2 which is the converse of Assertion 1. This will be shown as valid and show the trace where the plane crashes. Assertion 3,4,5 are used to find out the maximum number of planes landed or took off given a time constraint. Assertion 6,7,8 are LTL assertions to ensure that there are no resource violations. Assertion 9 is able to find the maximum amount of fuel used by a plane in the best scheduling sequence. Given the maximum fuel used, the minimum fuel required for each plane can be determined. These assertions will be further explored in detail in the experimental section.

2.2.2 Experiments and Discussion

Test cases are devised and using the different assertions above, different rounds of verifications are followed:

Round 1a: Using assertion 1 to verify if 3 planes can be landed without any plane crashing

Round 1b:

- Using assertion 1 to verify if 5 planes can be landed without any plane crashing
- Using assertion 2 to trace which plane has crashed and the scheduling sequence that

leads to it

Round 2:

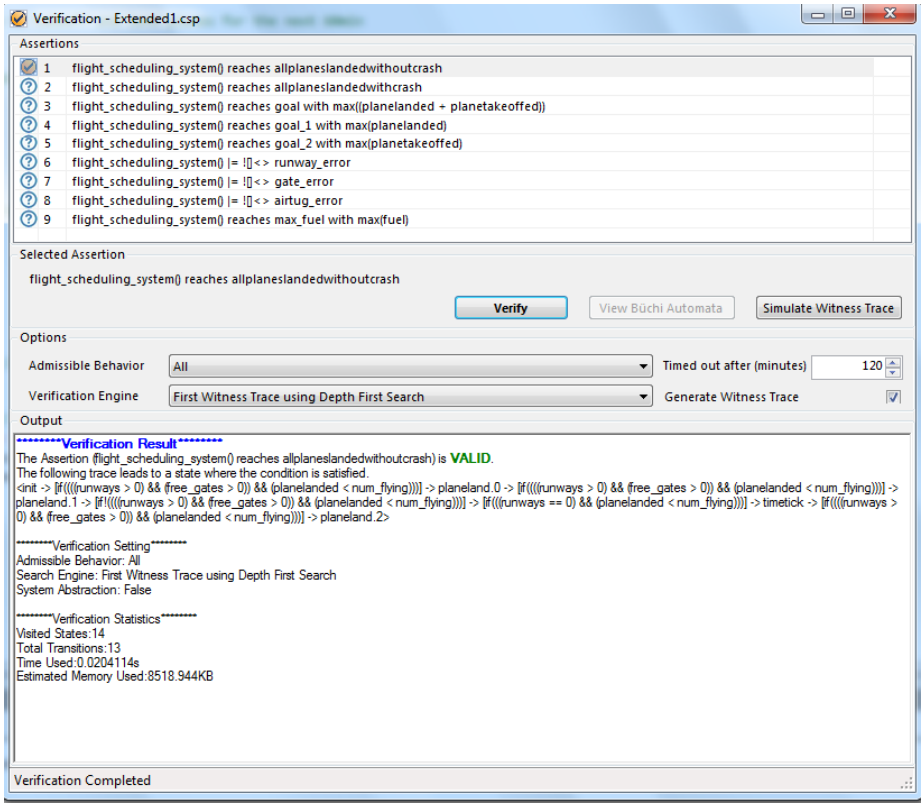
- Using assertion 3 to verify that this model proves the same 40 flight/hour figure with the added fuel, gate and airtug constraints.

- Using assertions 6, 7 and 8 to verify adherence to runway, gate and airtug constraints.

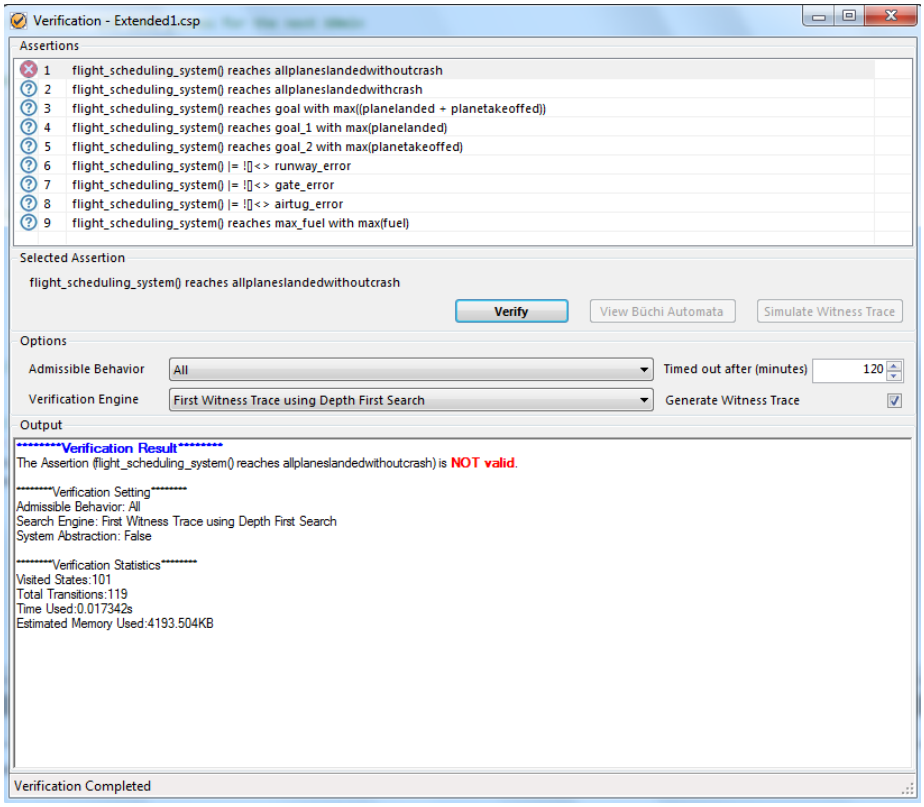
Round 3a & Round 3b: Using assertion 9 to find minimum fuel required for each plane.

Variable	Round 1a	Round 1b	Round 2	Round 3a	Round 3b
Planes	3	5	41	41	41
Runways	2	2	2	2	2
Gates	3	3	3	3	3
Tugs	3	3	3	3	3
Fuel	300	300	5000	5000	3985

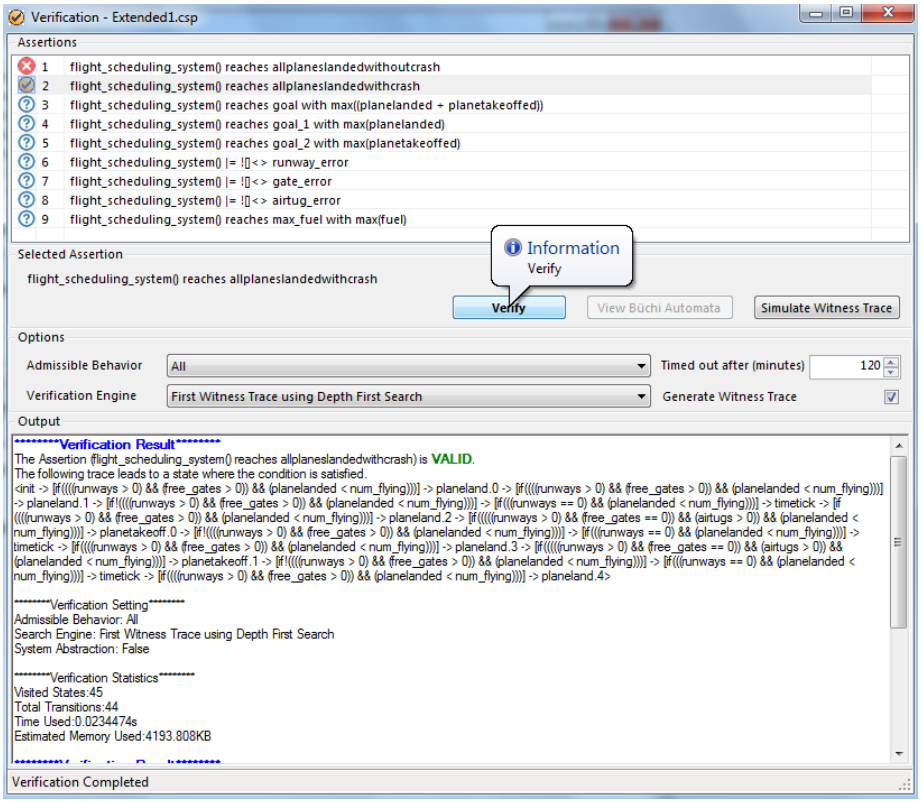
Running PAT following the above rounds of verifications, firstly, 3 planes can be landed without any planes crashing as shown below:



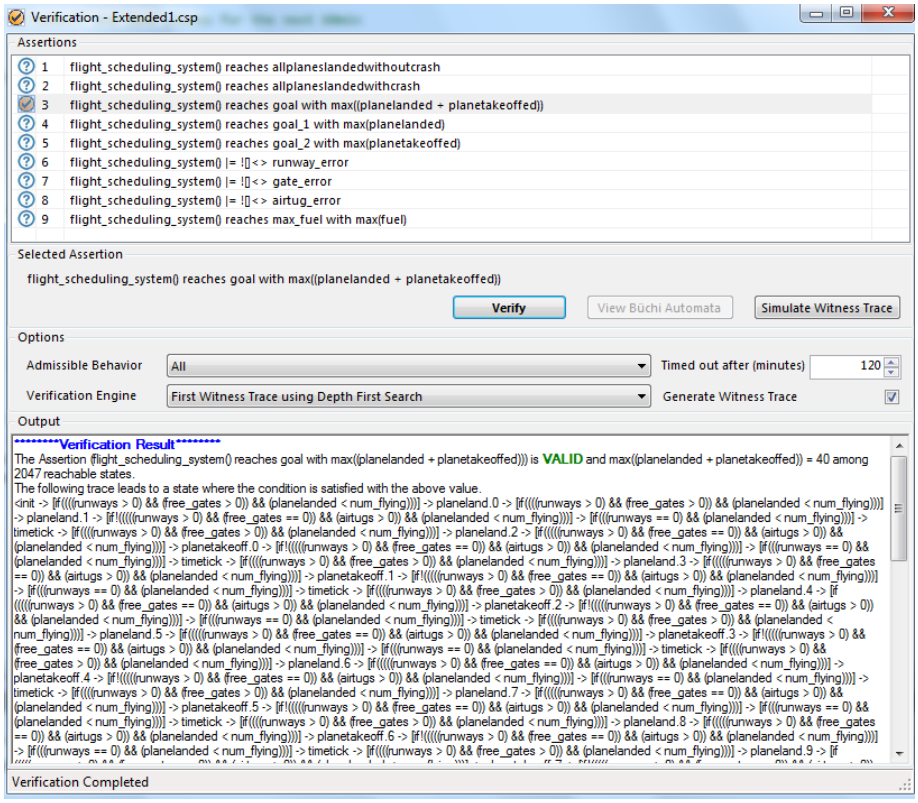
However, for the test case of 5 planes in Round 1b, not every plane landed successfully.



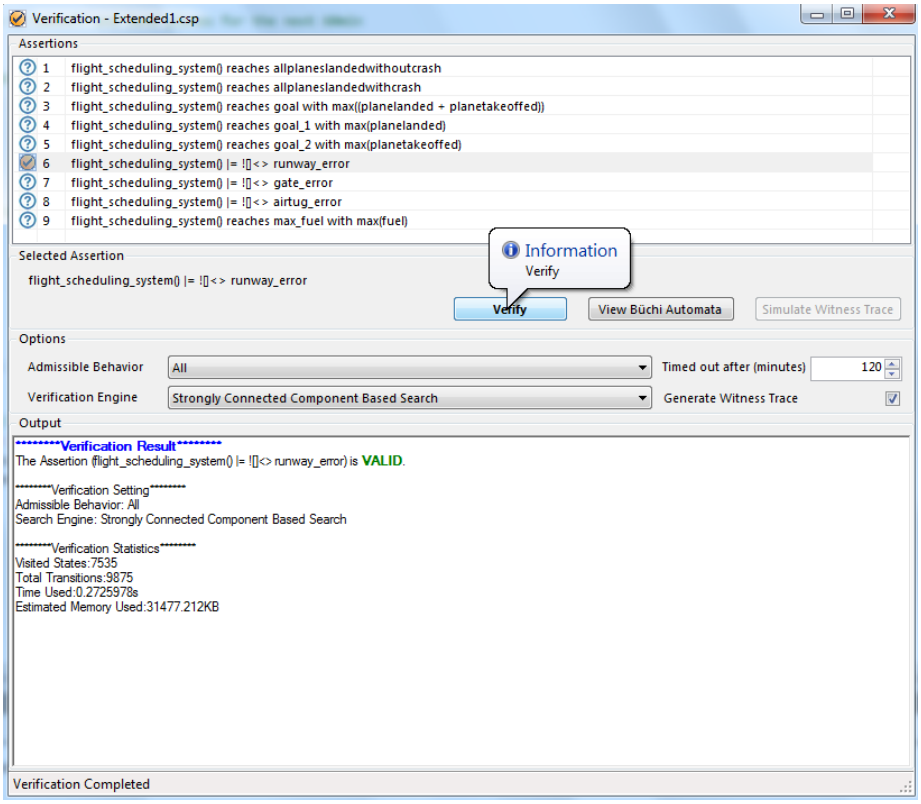
Running assertion 2 which is the contrary of assertion 1 on Round 1b allows us to trace and to find that plane 4 crashed.

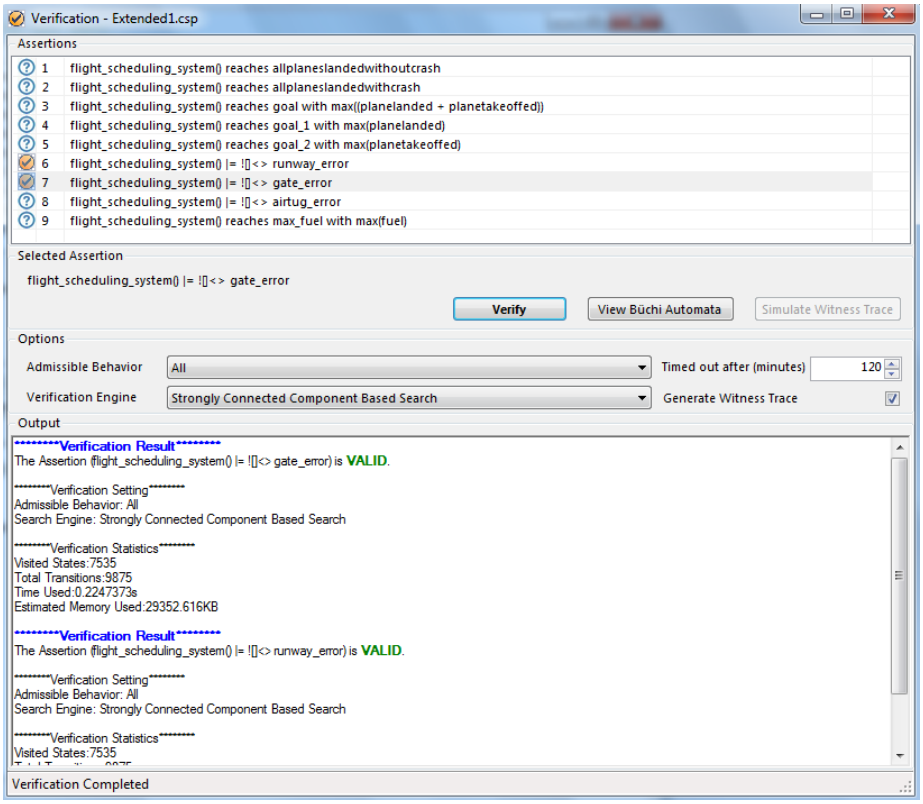


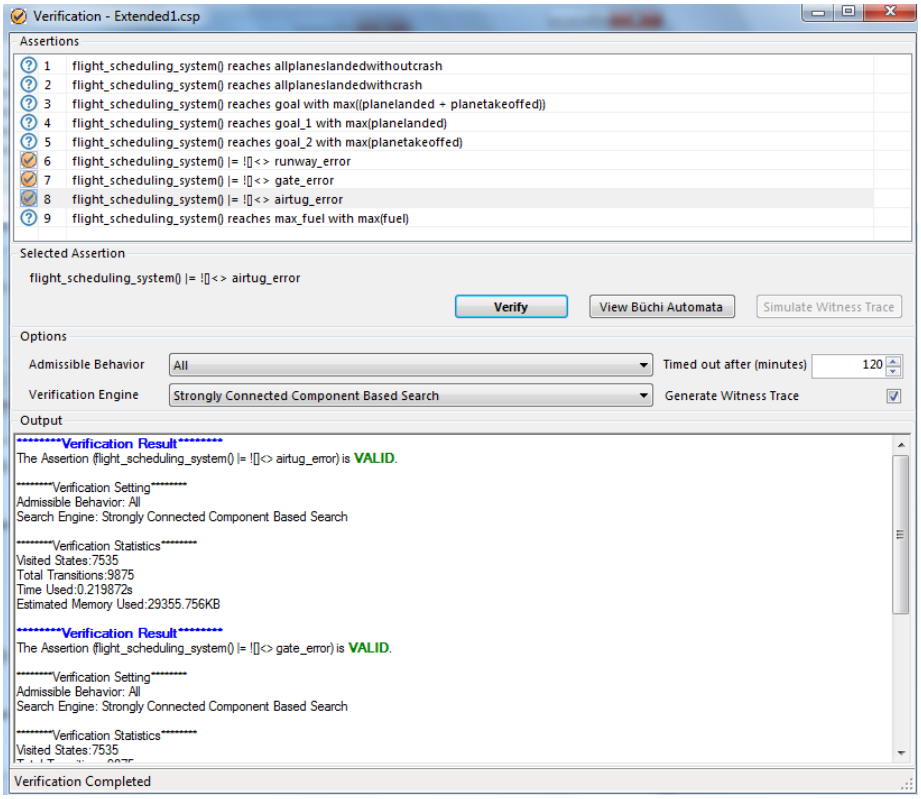
In round 2, since 5 planes will incur a plane-crashing event, 41 planes will definitely not be successful given the same resource setup. Hence, the fuel levels for all 41 planes were increased to an arbitrary large value of 5000. Again, running the PAT proves that in an hour, 40 flights can be handled given 2 runways.



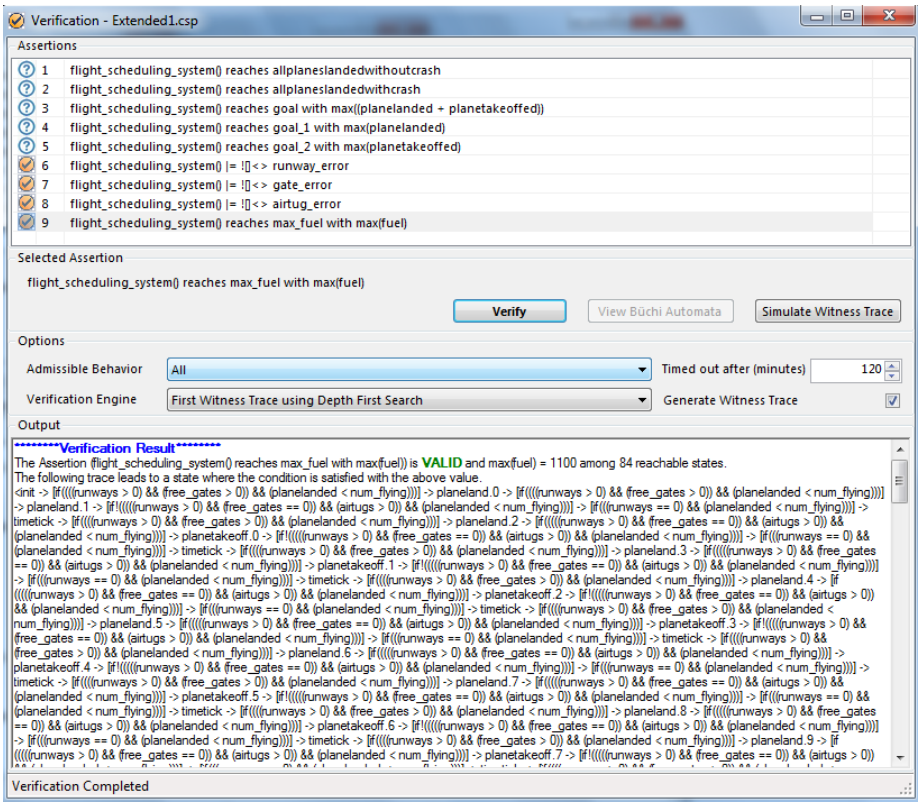
And assertions 6, 7 and 8 proved the adherence of the flight scheduling system in this model


Verification - Extended1.csp
Assertions
1 flight_scheduling_system() reaches allplaneslandedwithoutcrash
2 flight_scheduling_system() reaches allplaneslandedwithcrash
3 flight_scheduling_system() reaches goal with max((planelanded + planetakeoffed))
4 flight_scheduling_system() reaches goal_1 with max(planelanded)
5 flight_scheduling_system() reaches goal_2 with max(planetakeoffed)
6 flight_scheduling_system() |= ![]<> runway_error
7 flight_scheduling_system() |= ![]<> gate_error
8 flight_scheduling_system() |= ![]<> airtug_error
9 flight_scheduling_system() reaches max_fuel with max(fuel)
Selected Assertion
flight_scheduling_system() |= ![]<> runway_error
Information
Verify
Verify
View Büchi Automata
Simulate Witness Trace
Options
Admissible Behavior
All
Timed out after (minutes)
120
Verification Engine
Strongly Connected Component Based Search
Generate Witness Trace
Output
********Verification Result********
The Assertion (flight_scheduling_system() |= ![]<> runway_error) is VALID.
********Verification Setting********
Admissible Behavior: All
Search Engine: Strongly Connected Component Based Search
********Verification Statistics********
Visited States:7535
Total Transitions:9875
Time Used:0.2725978s
Estimated Memory Used:31477.212KB
Verification Completed


Verification - Extended1.csp
Assertions
1 flight_scheduling_system() reaches allplaneslandedwithoutcrash
2 flight_scheduling_system() reaches allplaneslandedwithcrash
3 flight_scheduling_system() reaches goal with max((planelanded + planetakeoffed))
4 flight_scheduling_system() reaches goal_1 with max(planelanded)
5 flight_scheduling_system() reaches goal_2 with max(planetakeoffed)
6 flight_scheduling_system() |= ![]<> runway_error
7 flight_scheduling_system() |= ![]<> gate_error
8 flight_scheduling_system() |= ![]<> airtug_error
9 flight_scheduling_system() reaches max_fuel with max(fuel)
Selected Assertion
flight_scheduling_system() |= ![]<> gate_error
Verify
View Büchi Automata
Simulate Witness Trace
Options
Admissible Behavior
All
Timed out after (minutes)
120
Verification Engine
Strongly Connected Component Based Search
Generate Witness Trace
Output
********Verification Result********
The Assertion (flight_scheduling_system() |= ![]<> gate_error) is VALID.
********Verification Setting********
Admissible Behavior: All
Search Engine: Strongly Connected Component Based Search
********Verification Statistics********
Visited States:7535
Total Transitions:9875
Time Used:0.2247373s
Estimated Memory Used:29352.616KB
********Verification Result********
The Assertion (flight_scheduling_system() |= ![]<> runway_error) is VALID.
********Verification Setting********
Admissible Behavior: All
Search Engine: Strongly Connected Component Based Search
********Verification Statistics********
Visited States:7535
Verification Completed

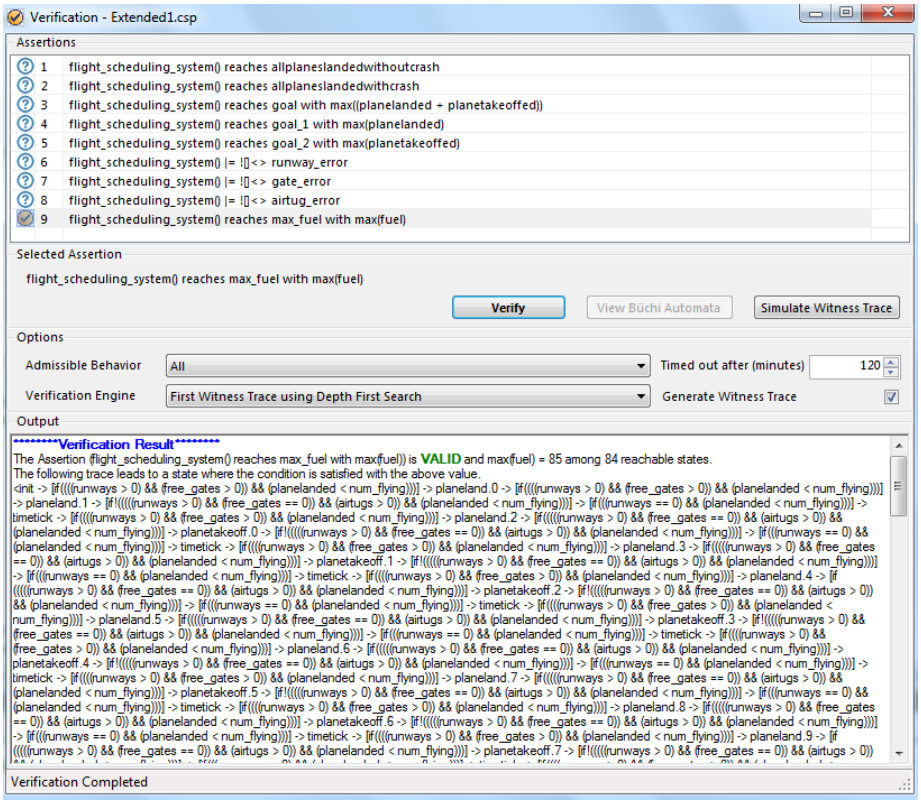

Verification - Extended1.csp
Assertions
1 flight_scheduling_system() reaches allplaneslandedwithoutcrash
2 flight_scheduling_system() reaches allplaneslandedwithcrash
3 flight_scheduling_system() reaches goal with max((planelanded + planetakeoffed))
4 flight_scheduling_system() reaches goal_1 with max(planelanded)
5 flight_scheduling_system() reaches goal_2 with max(planetakeoffed)
6 flight_scheduling_system() |= ![]<> runway_error
7 flight_scheduling_system() |= ![]<> gate_error
8 flight_scheduling_system() |= ![]<> airtug_error
9 flight_scheduling_system() reaches max_fuel with max(fuel)
Selected Assertion
flight_scheduling_system() |= ![]<> airtug_error
Verify
View Büchi Automata
Simulate Witness Trace
Options
Admissible Behavior
All
Timed out after (minutes)
120
Verification Engine
Strongly Connected Component Based Search
Generate Witness Trace
Output
********Verification Result********
The Assertion (flight_scheduling_system() |= ![]<> airtug_error) is VALID.
********Verification Setting********
Admissible Behavior: All
Search Engine: Strongly Connected Component Based Search
********Verification Statistics********
Visited States:7535
Total Transitions:9875
Time Used:0.219872s
Estimated Memory Used:29355.756KB
********Verification Result********
The Assertion (flight_scheduling_system() |= ![]<> gate_error) is VALID.
********Verification Setting********
Admissible Behavior: All
Search Engine: Strongly Connected Component Based Search
********Verification Statistics********
Visited States:7535
Verification Completed

Lastly, using assertion 9, the maximum fuel remained each plane after all planes have landed is identified using the same resource setup. The max(fuel) indicates that for 5000 fuel level for every of the 41 planes, the maximum fuel remained, equivalent to the remaining fuel in the last plane that landed is 1100.

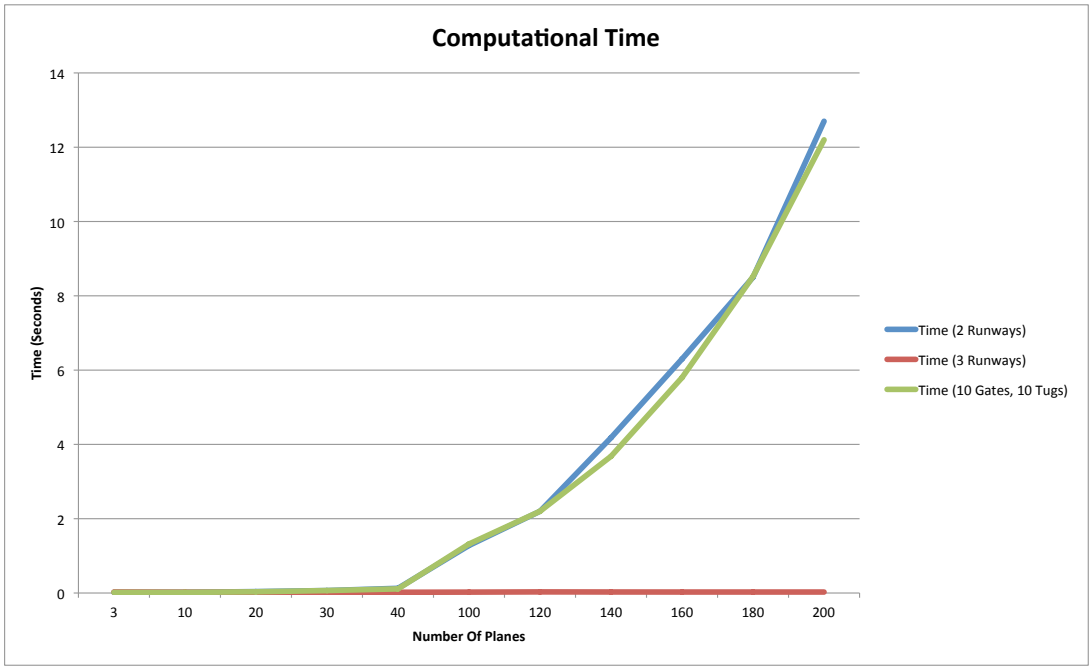


To find the minimum fuel: 5000-1100+85 =3985.

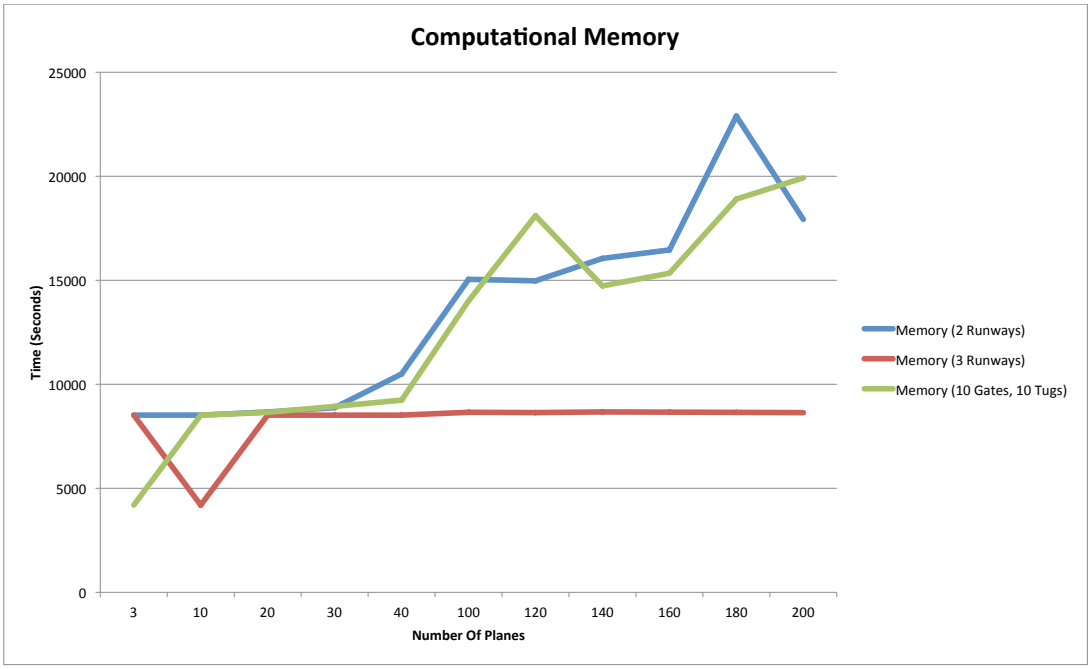
For a plane to land without crashing, the minimum fuel for all 41 planes to land successfully is 3985. We assign 3985 as fuel level for all 41 planes and the verification result is as shown:



3985 is the minimum fuel for all airplanes given that number of airplanes on land and flying. A verification shows an invalid result when 3984 is given for all airplanes. More verification attempts were made with alterations to resource figures to show the relationship of the number of planes with computation time and computation memory.



As each plane is an individual object, increasing the number of planes increases the state space and the number of transitions. This results in an exponential increase in computation time. When 2 runways was increased to 3, the number of transitions sharply decreased. There was almost no linear growth in computation time. However, modifying the number of gates and airtugs made no difference to the original computation time. This concludes that the runway is the main bottleneck of the model. Hence, Changi Airport can handle much more flights by adding more runways without having to increase the number of gates or air tugs.



As for computation memory, the same benefit can be seen when the runways was increased to 3 and increasing the number of gates and airtugs made no significant difference.

2.3 Tertiary Model

2.3.1 Model Description

In a scenario in an airport 2 runways and 3 gates with 5 airplanes in the air waiting to land, with the planes status being:

1. fuel=300, emergency=yes
2. fuel=200, emergency=no
3. fuel=100, emergency=yes
4. fuel=200, emergency=no
5. fuel=300, emergency=yes

the variables would be setup as such.

```
//Airport resource variables (user input)
#define num_runways 2;
#define num_gates 3;

//Airplanes variables (user input)
#define num_flying 5;
#define num_parked 0; //num_parked <= num_gates

//Process Variables (user input)
var planesflying[num_flying] = [plane_flying(num_flying)];
var fuel[num_flying] = [300,200,100,200,300]; //same number as num_flying
var emergency[num_flying] = [yes,no,yes,no,yes]; //same number as num_flying
var runways = num_runways;
var free_gates = num_gates-num_parked;

//Assertion variables
var timer=0;
var crash=0;
var planeslanded=0;
var index=0;
```

The system gives priority for plane with critically low fuel (less than 165 gallons) to land first before scheduling planes with emergency and finally normal flying planes. Gate constraint is given an exception for emergency or critically low fueled planes. Normal flying planes cannot land when there are no free gates while planes with emergency and critically low fueled planes will be allowed to land without free gates as long as the runway is available. It would be too late to wait for a gate to be freed as this will involve waiting for a parked plane to take off before a gate can be freed. Planes will crash if they run out of fuel. It takes 3 minutes for each use of the runway. Planes waiting to land will use 80 gallons of fuel while circling in the air for 3 minutes.

2.3.2 Experiments and Discussion

Running the system on PAT gives the following schedule:

**********Verification Result**********

```
The Assertion (flight_scheduling_system() reaches allplaneslandedwithoutcrash) is VALID.
The following trace leads to a state where the condition is satisfied.
<init -> [if((planeslanded < num_flying))] -> [if!(((index == num_flying) || (runways == 0)))] -> [if(((planesflying[index] !=
plane_flying) || ((planesflying[index] == plane_flying) && ((fuel[index] - 80) > 84))))] ->
τ -> [if((planeslanded < num_flying))] -> [if!(((index == num_flying) || (runways == 0)))] -> [if(((planesflying[index] !=
plane_flying) || ((planesflying[index] == plane_flying) && ((fuel[index] - 80) > 84))))] -> τ -> [if((planeslanded < num_flying))] ->
[if!(((index == num_flying) || (runways == 0)))] -> [if!(((planesflying[index] != plane_flying) || ((planesflying[index] ==
plane_flying) && ((fuel[index] - 80) > 84))))] -> [if(((((planesflying[index] == plane_flying) && (index < num_flying)) &&
((fuel[index] - 80) < 85)) && (runways > 0)))] -> FUELland.2 -> [if((planeslanded < num_flying))] -> [if!(((index == num_flying) ||
(runways == 0)))] -> [if(((planesflying[index] != plane_flying) || ((planesflying[index] == plane_flying) && ((fuel[index] - 80) >
84))))] -> τ -> [if((planeslanded < num_flying))] -> [if!(((index == num_flying) || (runways == 0)))] -> [if(((planesflying[index] !=
plane_flying) || ((planesflying[index] == plane_flying) && ((fuel[index] - 80) > 84))))] -> τ -> [if((planeslanded < num_flying))] ->
[if(((index == num_flying) || (runways == 0)))] -> τ -> [if((planeslanded < num_flying))] -> [if!(((index == num_flying) || (runways
== 0)))] -> [if!(((planesflying[index] != plane_flying) || ((planesflying[index] == plane_flying) && (emergency[index] == no))))] ->
[if(((((planesflying[index] == plane_flying) && (index < num_flying)) && (emergency[index] == yes)) && (runways > 0)))] ->
EMERGENCYland.0 -> [if((planeslanded < num_flying))] -> [if(((index == num_flying) || (runways == 0)))] -> τ ->
[if((planeslanded < num_flying))] -> [if(((index == num_flying) || (runways == 0)))] -> τ -> [if((planeslanded < num_flying))] ->
TAKEOFFnumber.0* -> minusfuel -> resetrunways -> [if((planeslanded < num_flying))] -> [if!(((index == num_flying) || (runways
== 0)))] -> [if(((planesflying[index] != plane_flying) || ((planesflying[index] == plane_flying) && ((fuel[index] - 80) > 84))))] -> τ ->
[if((planeslanded < num_flying))] -> [if!(((index == num_flying) || (runways == 0)))] -> [if!(((planesflying[index] != plane_flying) ||
((planesflying[index] == plane_flying) && ((fuel[index] - 80) > 84))))] -> [if(((((planesflying[index] == plane_flying) && (index <
num_flying)) && ((fuel[index] - 80) < 85)) && (runways > 0)))] -> FUELland.1 -> [if((planeslanded < num_flying))] -> [if!(((index
== num_flying) || (runways == 0)))] -> [if(((planesflying[index] != plane_flying) || ((planesflying[index] == plane_flying) &&
((fuel[index] - 80) > 84))))] -> τ -> [if((planeslanded < num_flying))] -> [if!(((index == num_flying) || (runways == 0)))] ->
[if!(((planesflying[index] != plane_flying) || ((planesflying[index] == plane_flying) && ((fuel[index] - 80) > 84))))] ->
[if(((((planesflying[index] == plane_flying) && (index < num_flying)) && ((fuel[index] - 80) < 85)) && (runways > 0)))] ->
FUELland.3 -> [if((planeslanded < num_flying))] -> [if(((index == num_flying) || (runways == 0)))] -> τ -> [if((planeslanded <
num_flying))] -> [if(((index == num_flying) || (runways == 0)))] -> τ -> [if((planeslanded < num_flying))] -> [if(((index ==
num_flying) || (runways == 0)))] -> τ -> [if((planeslanded < num_flying))] -> TAKEOFFnumber.0* -> minusfuel -> resetrunways ->
[if((planeslanded < num_flying))] -> [if!(((index == num_flying) || (runways == 0)))] -> [if(((planesflying[index] != plane_flying) ||
((planesflying[index] == plane_flying) && ((fuel[index] - 80) > 84))))] -> τ -> [if((planeslanded < num_flying))] -> [if!(((index ==
num_flying) || (runways == 0)))] -> [if(((planesflying[index] != plane_flying) || ((planesflying[index] == plane_flying) &&
((fuel[index] - 80) > 84))))] -> τ -> [if((planeslanded < num_flying))] -> [if!(((index == num_flying) || (runways == 0)))] ->
[if(((planesflying[index] != plane_flying) || ((planesflying[index] == plane_flying) && ((fuel[index] - 80) > 84))))] -> τ ->
[if((planeslanded < num_flying))] -> [if!(((index == num_flying) || (runways == 0)))] -> [if(((planesflying[index] != plane_flying) ||
((planesflying[index] == plane_flying) && ((fuel[index] - 80) > 84))))] -> τ -> [if((planeslanded < num_flying))] -> [if!(((index ==
num_flying) || (runways == 0)))] -> [if!(((planesflying[index] != plane_flying) || ((planesflying[index] == plane_flying) &&
((fuel[index] - 80) > 84))))] -> [if(((((planesflying[index] == plane_flying) && (index < num_flying)) && ((fuel[index] - 80) < 85))
&& (runways > 0)))] -> FUELland.4>
```

Takeoff number is determined based on the remaining number of runways available. If there are planes parked at the gate and a runway is available, a takeoff will be scheduled to free the gate for other planes to land. Fuel will be removed from the remaining planes when the runway are filled before scheduling the next cycle. Plane array start from 0. so the actual indicated planes would be +1.

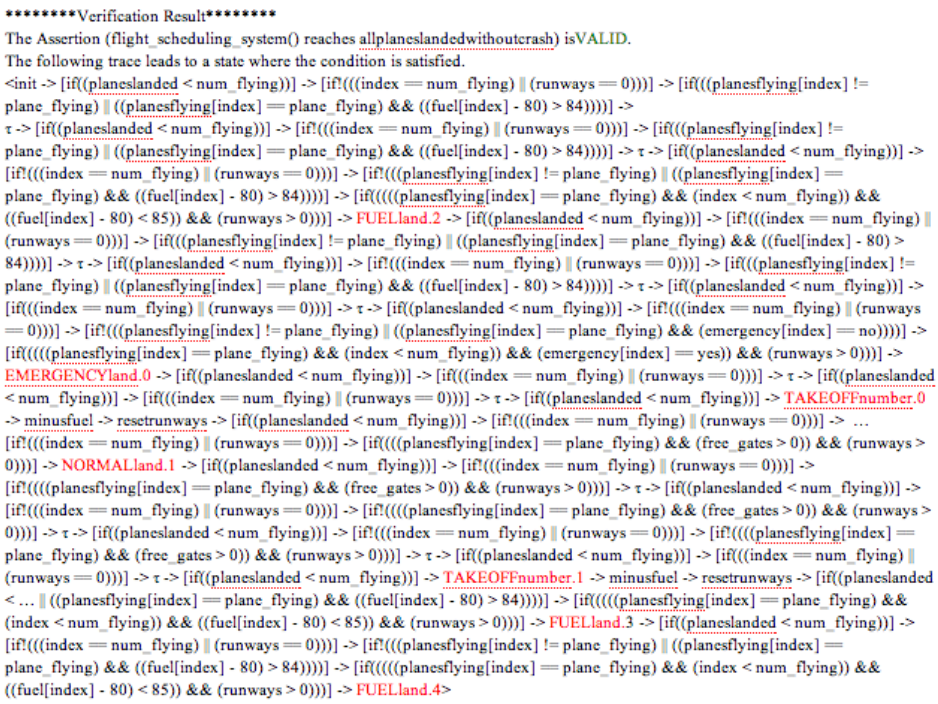
In Summary, the schedule would be:

Plane3(low fuel) and Plane1(emergency) will land → Flying planes fuel decrease → Plane2(low fuel) and Plane4(low fuel) will land → Flying planes fuel decrease → Plane5(low fuel) will land In a scenario in an airport 2 runways and 3 gates with 5 airplanes in the air waiting to land,

with the planes status being:

1. fuel=300, emergency=yes
2. fuel=300, emergency=no
3. fuel=100, emergency=yes
4. fuel=300, emergency=no
5. fuel=300, emergency=no

Running the system on PAT gives the following schedule:



In Summary, the schedule would be:

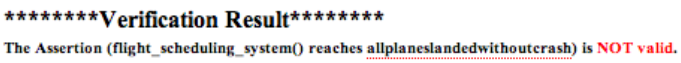
Plane3(low fuel) and Plane1(emergency) will land → Flying planes fuel decrease → Plane2 will land and 1 plane will takeoff → Flying planes fuel decrease → Plane4(low fuel) and Plane5(low fuel) will land. Plane4 and Plane5 have to wait till there is a free gate (takeoff) before the plane can be scheduled to land. (Gate constraint)

In a scenario in an airport 2 runways and 3 gates with 5 airplanes in the air waiting to land,

with the planes status being:

1. fuel=200, emergency=yes
2. fuel=200, emergency=no
3. fuel=200, emergency=yes
4. fuel=200, emergency=no
5. fuel=200, emergency=no

Running the system on PAT gives the following schedule:



All the planes have too little fuel and cannot be scheduled to land on 2 runways such that they will not crash (run out of fuel). A possible crash scenario is shown using PAT:

********Verification Result********

The Assertion (flight_scheduling_system() reaches allplaneslandedwithcrash) is **VALID**.

The following trace leads to a state where the condition is satisfied.

<init -> [if((planeslanded < num_flying))] -> [if!(((index == num_flying) || (runways == 0)))] -> [if(((planesflying[index] != ...
[if(((((planesflying[index] == plane_flying) && (index < num_flying)) && (emergency[index] == yes)) && (runways > 0)))] ->
EMERGENCYland.0 -> [if((planeslanded < num_flying))] -> [if!(((index == num_flying) || (runways == 0)))] ->
[if(((planesflying[index] != plane_flying) || ((planesflying[index] == plane_flying) && (emergency[index] == no))))] -> τ ->
[if((planeslanded < num_flying))] -> [if!(((index == num_flying) || (runways == 0)))] -> [if!(((planesflying[index] != plane_flying) ||
((planesflying[index] == plane_flying) && (emergency[index] == no)))))] -> [if(((((planesflying[index] == plane_flying) && (index
< num_flying)) && (emergency[index] == yes)) && (runways > 0)))] -> EMERGENCYland.2 -> [if((planeslanded < num_flying))]
-> [if(((index == num_flying) || (runways == 0)))] -> τ -> [if((planeslanded < num_flying))] -> [if(((index == num_flying) || (runways
== 0)))] -> τ -> [if((planeslanded < num_flying))] -> TAKEOFFnumber.0 -> minusfuel -> resetrunways -> [if((planeslanded <
num_flying))] -> [if!(((index == num_flying) || (runways == 0)))] -> [if(((planesflying[index] != plane_flying) || ((planesflying[index]
== plane_flying) && ((fuel[index] - 80) > 84)))))] -> τ -> [if((planeslanded < num_flying))] -> [if!(((index == num_flying) ||
(runways == 0)))] -> [if!(((planesflying[index] != plane_flying) || ((planesflying[index] == plane_flying) && ((fuel[index] - 80) >
84)))))] -> [if(((((planesflying[index] == plane_flying) && (index < num_flying)) && ((fuel[index] - 80) < 85)) && (runways > 0)))]
-> FUELland.1 -> [if((planeslanded < num_flying))] -> [if!(((index == num_flying) || (runways == 0)))] -> [if(((planesflying[index]
!= plane_flying) || ((planesflying[index] == plane_flying) && ((fuel[index] - 80) > 84)))))] -> τ -> [if((planeslanded < num_flying))]
-> [if!(((index == num_flying) || (runways == 0)))] -> [if!(((planesflying[index] != plane_flying) || ((planesflying[index] ==
plane_flying) && ((fuel[index] - 80) > 84)))))] -> [if(((((planesflying[index] == plane_flying) && (index < num_flying)) &&
((fuel[index] - 80) < 85)) && (runways > 0)))] -> FUELland.3 -> [if((planeslanded < num_flying))] -> [if(((index == num_flying) ||
(runways == 0)))] -> τ -> [if((planeslanded < num_flying))] -> [if(((index == num_flying) || (runways == 0)))] -> τ ->
[if((planeslanded < num_flying))] -> [if(((index == num_flying) || (runways == 0)))] -> τ -> [if((planeslanded < num_flying))] ->
TAKEOFFnumber.0 -> minusfuel -> resetrunways -> [if((planeslanded < num_flying))] -> [if!(((index == num_flying) || (runways ==
0)))] -> [if(((planesflying[index] != plane_flying) || ((planesflying[index] == plane_flying) && ((fuel[index] - 80) > 84)))))] -> τ ->
[if((planeslanded < num_flying))] -> [if!(((index == num_flying) || (runways == 0)))] -> [if(((planesflying[index] != plane_flying) ||
((planesflying[index] == plane_flying) && ((fuel[index] - 80) > 84)))))] -> τ -> [if((planeslanded < num_flying))] -> [if!(((index ==
num_flying) || (runways == 0)))] -> [if(((planesflying[index] != plane_flying) || ((planesflying[index] == plane_flying) &&
((fuel[index] - 80) > 84)))))] -> τ -> [if((planeslanded < num_flying))] -> [if!(((index == num_flying) || (runways == 0)))] ->
[if(((planesflying[index] != plane_flying) || ((planesflying[index] == plane_flying) && ((fuel[index] - 80) > 84)))))] -> τ ->
[if((planeslanded < num_flying))] -> [if!(((index == num_flying) || (runways == 0)))] -> [if!(((planesflying[index] != plane_flying) ||
((planesflying[index] == plane_flying) && ((fuel[index] - 80) > 84)))))] -> [if(((((planesflying[index] == plane_flying) && (index <
num_flying)) && ((fuel[index] - 80) < 85)) && (runways > 0)))] -> FUELland.4>

In Summary, the crash would be:

Plane3(emergency) and Plane1(emergency) will land → Flying planes fuel decrease → Plane2(low fuel) and Plane4(low fuel) will land → Flying planes fuel decrease → Plane5(run out of fuel) (200-80-80=40) < 85.

2.4 Extended Tertiary Model

2.4.1 Model Description

In this last extended version of the tertiary model, the system uses sorted queues to land awaiting planes in the airspace.This model sorts all flying planes firstly by emergency needs followed by fuel levels since fuel has a priority over emergencies. In addition, since this model sorts before scheduling planes to land, the fuel threshold in the previous tertiary model has been removed to avoid instantaneous landing before completion of the

sorting processes.

2.4.2 Experiments and Discussion

In the first case, we show that fuel priority has higher precedence than emergency priority. The scenario is set as 2 runways, 3 gates with 8 airplanes in the airspace waiting to land:

Plane 0 : fuel=100, emergency=no

Plane 1: fuel=900, emergency = no

Plane 2: fuel=900, emergency = no

Plane 3: fuel=900, emergency = no

Plane 4: fuel=900, emergency = no

Plane 5: fuel=900, emergency = no

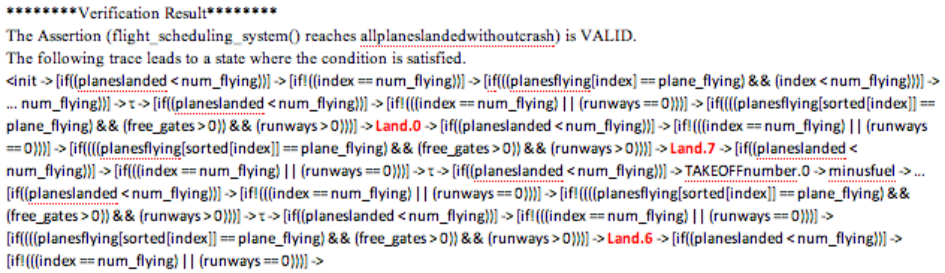
Plane 6: fuel=900, emergency = yes

Plane 7: fuel=200, emergency = no

After both emergency and fuel sorting processes, the ideal schedule should be as such:

Plane 0 → Plane 7 → Plane 6 → The Rest

After running PAT with the assertion that the flight scheduling system finds a schedulable algorithm to land all 8 planes without crashing, the result tallies with the ideal schedule above.



In the second case, we show a scenario where scheduling deals with fuel priority only. The scenario is set as 2 runways, 3 gates with 8 airplanes in the airspace waiting to land:

Plane 0 : fuel=200, emergency=no

Plane 1: fuel=100, emergency = no

Plane 2: fuel=400, emergency = no

Plane 3: fuel=300, emergency = no

Plane 4: fuel=800, emergency = no

Plane 5: fuel=700, emergency = no

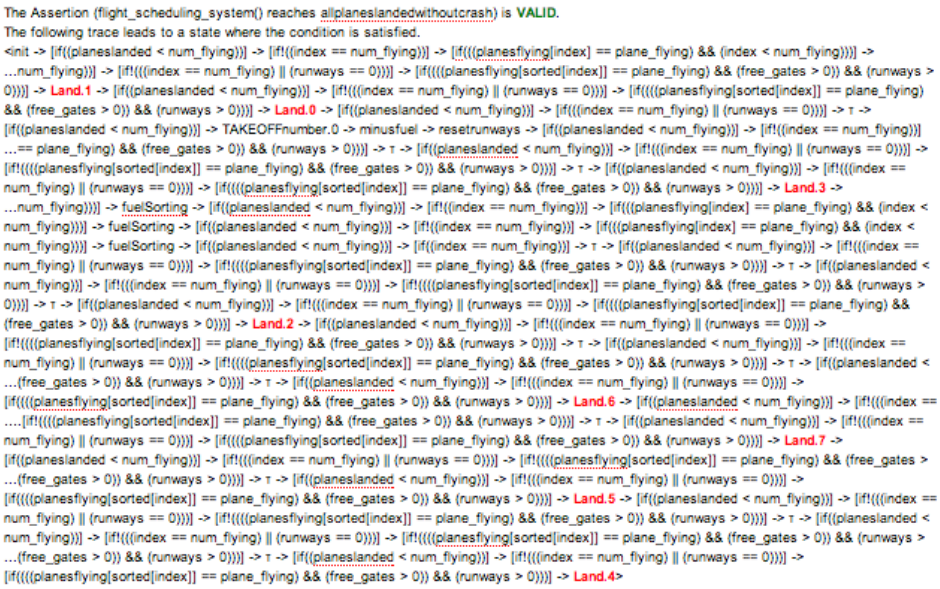
Plane 6: fuel=500, emergency = no

Plane 7: fuel=600, emergency = no

After both emergency and fuel sorting processes, the ideal schedule should be as such:

Plane 1 $\rightarrow$ Plane 0 $\rightarrow$ Plane 3 $\rightarrow$ Plane 2 $\rightarrow$ Plane 6 $\rightarrow$ Plane 7 $\rightarrow$ Plane 5 $\rightarrow$ Plane 4

After running PAT with the assertion that the flight scheduling system finds a schedulable algorithm to land all 8 planes without crashing, the result tallies with the ideal schedule above.



Last but not least in the third case, we show a model that deals with emergency priorities only. The scenario is set as 2 runways, 3 gates with 8 airplanes in the airspace waiting to land:

Plane 0 : fuel=900, emergency=no

Plane 1: fuel=900, emergency = no

Plane 2: fuel=900, emergency = yes

Plane 3: fuel=900, emergency = yes

Plane 4: fuel=900, emergency = no

Plane 5: fuel=900, emergency = no

Plane 6: fuel=900, emergency = yes

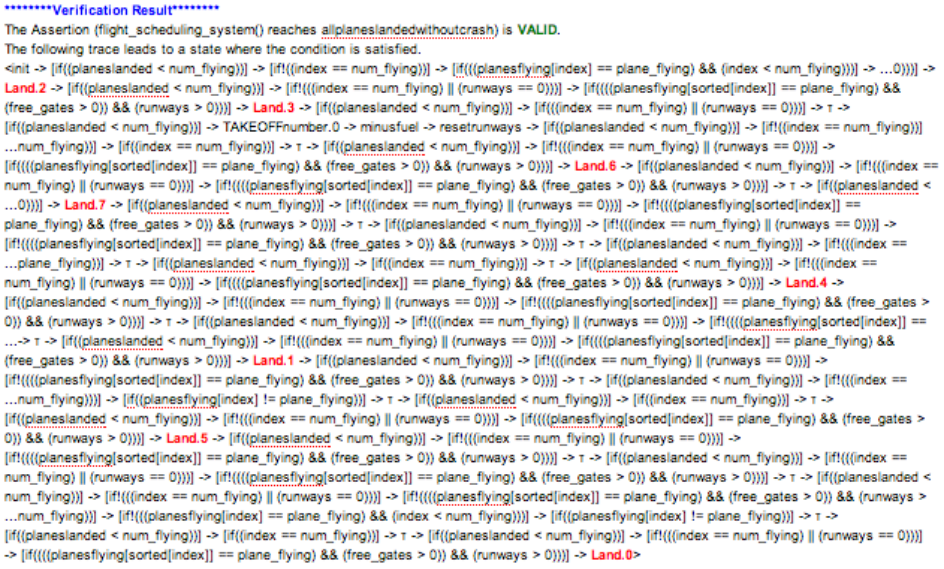
Plane 7: fuel=900, emergency = yes

After both emergency and fuel sorting processes, the ideal schedule should be as such:

Plane 2 → Plane 3 → Plane 6 → Plane 7 → Plane 4 → Plane 1 → Plane 5 → Plane 0

After running PAT with the assertion that the flight scheduling system finds a schedu-

lable algorithm to land all 8 planes without crashing, the result tallies with the ideal schedule above.



Chapter 3

Limitations

1. Elements in array are static.

Dynamic handling like using arrays as a stack (push, pop) or queue (enqueue, dequeue) and dynamically increasing array size is not supported. However, we have managed to (in the priority model) iterate through an array.

2. Difficulties faced in modelling concurrent systems.

PAT supports mutable shared variables and asynchronous channels. We have experimented with implementing semaphores and atomic regions. However, our difficulty is in understanding how each process is called. The Simulation is limited in how it can display processes in parallel.

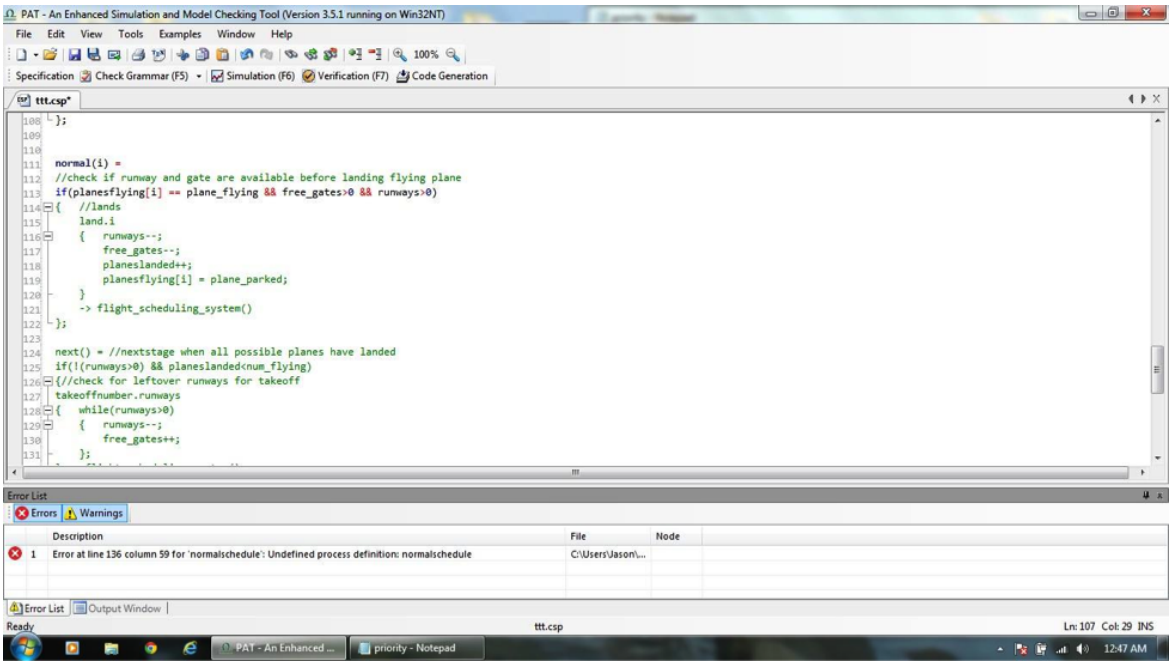
3. More objects requires an exponentially larger state space.

This will lead to an overwhelming computational complexity. We have implemented three separate models to manage the large complexity of the system that we are modelling.

Chapter 4

PAT Feedback

Feedback - Based on PAT 3.5.1 64-bit



1. Code segments are commented away undesirably.

2. The process that has its name in the lowest alphabetical order is simulated despite it not being called in the process flow statement.

Suggestion-wise, it would be good to include a feature to monitor selected variables (in a popup or in the sidebar separately) as the trace is simulated for a better debugging experience.

Chapter 5

Conclusion

There are many ways to model the flight scheduling system using PAT. PAT is a useful tool for supporting such real-time systems. Through the project, the user-friendly environment of PAT allowed the model checking of the flight scheduling system design to be done easily. The visual trace produced helps to promote a better understanding of the states and transitions.

Based on the verification results from PAT, the relationship between runways and flights capacity was investigated and the best flight schedule was obtained. By reducing the number of processes in the model, PAT worked better and provided a more efficient answer to real-world problems. This is an excellent alternative to creating finite numbers of tests to do quality assurance on a system model.

If given more time, PAT could be used to model a larger aspect of the flight scheduling system to come out with a more comprehensive flight scheduling solution. Overall, it was a pleasure using PAT. The syntax and logic was reasonably fair to pick up and we will continue to use PAT in the system verification of other projects.

References

[1] Gregory Karp, "Air travel to nearly double in next 20 years", 2012. (http://articles.chicagotribune.com/2012-03-08/business/chi-air-travel-to-nearly-double-in-next-20-years-faa-says-20120308_1_air-travel-air-traffic-forecasts)

[2] "Changi reports 9.4% rise in passenger traffic in August", 2013. (http://www.airport-technology.com/news/newschangi-reports-rise-passenger-traffic-august)

[3] Sun, Jun, et al, "PAT: Towards flexible verification under fairness." Computer Aided Verification, Springer Berlin Heidelberg, 2009.