# ASSESSMENT A1

|  |
| --- |
| 1. What is Bisimulation, and how can it be applied to verify the equivalence of two processes? Write a C++ program demonstrating Bisimulation between two processes. |
| Bisimulation is used to compare if 2 processes are behaviourally equivalent. If these states perform the same action but transition to the same state anyway, that is being bisimilar, and this is how equivalence of 2 processes are checked. Formally, simulations can be simulated via Labelled Transition Systems (L). For a given binary relation R with a set of states L, R is a bisimulation if a state x and transition to x’, and if there is a state y that can transition to y’, (x’, y’) exists in R as (x,y) also exists in R by default. This rule applies vice versa, where if there is a state x’ and y’ that can transition to x and y respectively, (x,y) must exist in R as (x’, y’) exist by default. This captures equivalence between states.  Below is the code on how bisimilarity can be shown, with states and inputs. If both processes end up in the same state and the same result under the condition of the same inputs, it shows bisimilarity in this bisimulation. In this example process X and Y are not bisimilar as they both do not have the same end result given the same input.   |  | | --- | | #include <iostream>  #include <unordered\_set>  using namespace std;  enum State { //states within this system  S0, //init state  S1,  S2,  numStates //number of states  };  enum Input { //inputs to transition between states  X,  Y,  numInputs  };  // transition for x input  State transitionX(State currState, Input currInput) {  switch (currState) {  case S0: //if input is X, then output is S1, else S2  return (currInput == X) ? S1 : S2;  case S1:  return (currInput == X) ? S2 : S1;  case S2:  return (currInput == X) ? S1 : S0;  default:  return S0;  }  }  // transition for y input  State transitionY(State currState, Input currInput) {  switch (currState) {  case S0:  return (currInput == X) ? S1 : S2;  case S1:  return (currInput == X) ? S2 : S1;  case S2: //diff with transitionX - when input is X, then transition should be to S1, else S0  return (currInput == X) ? S2 : S0;  default:  return S0;  }  }  // if transitions, no matter the input, can get the same results, will be bisimilar  bool isBisimilar() {  for (int s = 0; s < numStates; ++s) { //loop through S0 ~ S2  for (int i = 0; i < numInputs; ++i) { //loop through X and Y  State x = transitionX((State)s, (Input)i);  State y = transitionY((State)s, (Input)i);    cout<<"Transition from State: "<<s<<"; input "<<i<<"\n";  cout<<"Transition X: "<< x << ", Transition Y: "<<y<<endl;    if (x != y) {  return false;  }  }  }  return true;  }  int main() {  if (isBisimilar()) {  cout << "Process X and Y are bisimilar.\n";  } else {  cout << "Process X and Y are not bisimilar.\n";  }  return 0;  } | |

|  |
| --- |
| 1. What is simulation in discrete event systems, and how can you represent it in C++? Write a basic simulation of a queue system where entities enter and exit over time |
| In a discrete event system, it models the behaviour and performance of processes occurring overtime. There exists states and transitions between them will only happen if events occur, and before said events take place, exists a period when the system ‘waits’ until the duration ends to transition to the next state. These events can trigger upon meeting certain conditions. Examples of these include scheduling tasks or queuing them, such as customers waiting for their turn to be served one after another. In C++, we can use the library ‘queue’ to simulate a real-life queue of customers, and use ‘random’ to generate random entry and exit times for each customer as they come and go.  Below is a simulation where there is a queue, and entities arrive and leave at random times.   |  | | --- | | #include <iostream>  #include <queue>  #include <random>  #include <chrono>  #include <thread>  // constexpr - variable evaluated at compile time  constexpr double ARRIVAL\_RATE = 0.5; // Mean arrival rate (entities per time unit)  constexpr double SERVICE\_RATE = 0.7; // Mean service rate (entities per time unit)  constexpr int SIMULATION\_TIME = 20; // Total simulation time (time units)  double exponential\_random(double rate) { //generate random exponential number  std::default\_random\_engine gen(std::random\_device{}()); //random number engine  //dist - expoentital distribution obj - can generate random num using 'rate'  std::exponential\_distribution<> dist(rate);  return dist(gen);  }  class QueueSystem {  public:  QueueSystem(double arrival\_rate, double service\_rate)  : arrival\_rate\_(arrival\_rate), service\_rate\_(service\_rate) {}  void run() { //loop the entities until simulation time ends  for (int time = 0; time < SIMULATION\_TIME; ++time) {  // Entity arrival  if (exponential\_random(arrival\_rate\_) < 1) {  queue\_.push(time);  std::cout << "Entity arrived at time " << time << std::endl;  }  // Entity departure  if (!queue\_.empty() && exponential\_random(service\_rate\_) < 1) {  int departure\_time = queue\_.front();  queue\_.pop();  std::cout << "Entity departed at time " << time << std::endl;  }  // Sleep for some time unit to simulate the passage of time  std::this\_thread::sleep\_for(std::chrono::milliseconds(100));  }  }  private: //declaring arrival rate, service / wait rate, queue  double arrival\_rate\_;  double service\_rate\_;  std::queue<int> queue\_;  };  int main() {  QueueSystem queue\_system(ARRIVAL\_RATE, SERVICE\_RATE);  queue\_system.run();  return 0;  } | |
| Below is a LTS of an example of a queue system:  A diagram of a diagram  Description automatically generated |

|  |
| --- |
| 1. How can Bisimulation be applied to verify the equivalence of two concurrent processes? Implement a C++ program with two concurrent processes and demonstrate their bisimilarity. |
| Bisimulation can verify the equivalence of two concurrent processes by capturing the relation between the two processes, and if the states and their transitions are similar no matter the input given.  The code below shows 2 processes running concurrently and determines if the processes are bisimilar.   |  | | --- | | #include <iostream>  #include <thread>  #include <mutex>  #include <vector>  using namespace std;  mutex mtx; //mutex to ensure proper sync  int x = 0; //process 1  int y = 0; //process 2  bool done = false; //boolean to stop the while loop  void process1() {  while (!done) {  mtx.lock();  cout << "Process P1: " << x << endl;  x++;  mtx.unlock();  this\_thread::sleep\_for(chrono::milliseconds(500)); // thread waits for 0.5 seconds  }  }  void process2() {  while (!done) {  mtx.lock();  cout << "Process P2: " << y << std::endl;  y++;  mtx.unlock();  this\_thread::sleep\_for(chrono::milliseconds(500)); // thread waits for 0.5 seconds  }  }  int main() {  thread t1(process1); //threading both processes  thread t2(process2);  vector<int> historyProcess1; //store integers from both processes  vector<int> historyProcess2;  // records outputs of both processes for a finite number of steps  for (int i = 0; i < 5; ++i) {  this\_thread::sleep\_for(chrono::milliseconds(1000));  mtx.lock();  historyProcess1.push\_back(x);  historyProcess2.push\_back(y);  mtx.unlock();  }  // compare outputs  bool bisimilar = true;  for (int i = 0; i < historyProcess1.size(); ++i) {  if (historyP1[i] != historyProcess2[i]) {  bisimilar = false;  break;  }  }  done = true; // stop the processes  if (bisimilar) {  cout << "Processes P1 and P2 are bisimilar." << endl;  } else {  cout << "Processes P1 and P2 are not bisimilar." << endl;  }  t1.join(); //join threads together  t2.join();  return 0;  } | |

# ASSESSMENT A2

|  |
| --- |
| 1. What is the concept of concurrent processes, and how does it differ from sequential processes? Write a C++ program with two concurrent processes that demonstrate parallel execution. |
| Concurrent processes are computations that are executed at the same time, and they can potentially interact with one another. Sequential processes are the opposite, where tasks are executed one after another sequentially. One difference is concurrent processes are executed in parallel on multicore processors, and the concurrency does involve controlling resources such that each process can be executed without conflict or too much delay. This is different from sequential processes as each process is guaranteed resources as each one is executed one by one. Concurrent processes are controlled by threads, which is where interaction between processes start from, sending signals to each other. One such signal is signal and wait, where a thread signals another thread and then waits until the response is received, implying synchronisation.  Below is a C++ program to demonstrate parallel execution of 2 concurrent processes. It uses the concept of threads, where 2 processes are created for execution. Process 1 and 2 are joined together, each having its own thread and executing at the same time. Sometimes process 2 executes first because the processor can decide to allocate time for process 2 first. This shows the nature of concurrency, where threads can execute depending on the processor’s allocated time for each thread.  #include <iostream>  #include <thread>  using namespace std;  // Process 1  void process1() {  for (int i = 5; i > 0; i--) {  cout << "Process 1 executing iteration " << i << endl;  }  }  // Process 2  void process2() {  for (int i = 5; i > 0; i--) {  cout << "Process 2 executing iteration " << i << endl;  }  }  int main() {  // Two threads for each process  thread t1(process1);  thread t2(process2);  // Joining threads together for execution  t1.join();  t2.join();  cout << "Both processes have finished execution." << endl;  return 0;  } |

|  |
| --- |
| 1. Illustrate a common concurrency issue in C++ and propose a solution |
| One common concurrency issue is when a race condition happens. This is when no proper synchronization happens when multiple threads can access and modify shared data. This can sometimes cause inaccuracy and incorrect behaviour and results.  For example, if there are 2 threads accessing a function with a for loop, where a number is being incremented 100 times by the same 2 threads, the first thread may be increment at the same time as the second thread, or when the first thread has already moved on but the second thread does not get the updated information. Thus, the end result can be 190 even though both threads have finished running 100 times, or 200 times in total.  Below is an example of code that can result in an inaccurate result:   |  | | --- | | #include <iostream>  #include <thread>  using namespace std;  // Counter - is shared by the threads  int counter = 0;  // Simple for loop func  void increment\_counter() {  for (int i = 0; i < 20000; ++i) {  counter++; // Increment counter without synchronization  }  }  int main() {  // 2 threads created  thread t1(increment\_counter);  thread t2(increment\_counter);  t1.join();  t2.join();  // Expected final value of the counter  int expected\_final\_value = 2 \* 20000;  // Print expected and observed final values of the counter  cout << "Expected final counter value: " << expected\_final\_value << endl;  cout << "Observed final counter value: " << counter << endl;  // Check if there was a race condition  if (counter != expected\_final\_value) {  cout << "Race condition detected!" << endl;  } else {  cout << "No race condition detected." << endl;  }  return 0;  } |   And below is an example of an inaccurate result:    Thus, to avoid this situation, there needs to be something to ensure proper synchronisation. One such method is through mutex, which ensures shared resources are not modified when it should not be so. It locks the shared resource when a thread accesses it, and until that thread is finished running that task, it remains locked and inaccessible by other threads, ensuring up to date and correct values for the next thread to run it.   |  | | --- | | #include <iostream>  #include <thread>  #include <mutex>  using namespace std;  // Counter - is shared by the threads  int counter = 0;  std::mutex mtx; //mutex for synchronization  // Simple for loop func  void increment\_counter() {  for (int i = 0; i < 20000; ++i) {  mtx.lock(); //lock the access to counter  counter++; // Increment counter without synchronization  mtx.unlock(); //make counter available again  }  }  int main() {  // 2 threads created  thread t1(increment\_counter);  thread t2(increment\_counter);  t1.join();  t2.join();  // Expected final value of the counter  int expected\_final\_value = 2 \* 20000;  // Print expected and observed final values of the counter  cout << "Expected final counter value: " << expected\_final\_value << endl;  cout << "Observed final counter value: " << counter << endl;  // Check if there was a race condition  if (counter != expected\_final\_value) {  cout << "Race condition detected!" << endl;  } else {  cout << "No race condition detected." << endl;  }  return 0;  } | |

# ASSESSMENT A3

# DESCRIPTION

The simulation created is a traffic simulation system using Java and JFrame, where there are traffic lights, cars, and pedestrians on the road. The cars that can be added are ‘Kancil’ and ‘Four Wheel’. The cars obey traffic rules, where they stop at red lights and slow down at yellow lights, and give way for pedestrians.

There are also real time statistics which can calculate the total number of vehicles and pedestrians, the traffic flow, traffic density and the average speed of the ‘road’.

This system was built using Java as the backend, and JFrame as the front end. The libraries used were AWT and swing, and the IDE used was IntelliJ. To run this code, go to the main class Traffic.java and run the main function.

The video demonstration will be provided in the zip file accompanying the coursework upload, and here is the link to the sotonac sharepoint video: [traffic-management-systems.mp4](https://sotonac.sharepoint.com/:v:/t/selfrecording404/EUBciavfh_lBgh5Uo976R_IB01ymeYXLh5HxQ8xzlck3kg?nav=eyJyZWZlcnJhbEluZm8iOnsicmVmZXJyYWxBcHAiOiJTdHJlYW1XZWJBcHAiLCJyZWZlcnJhbFZpZXciOiJTaGFyZURpYWxvZy1MaW5rIiwicmVmZXJyYWxBcHBQbGF0Zm9ybSI6IldlYiIsInJlZmVycmFsTW9kZSI6InZpZXcifX0%3D&email=M.I.Babar%40soton.ac.uk&e=H0ShMW).

# IMPLEMENTATION

A parent class ‘Vehicle’ was created so that child classes ‘Kancil’ and ‘Four Wheel’ can be created, as each car has its own dimensions – width and height, and speed, which is the amount of spaces a car can travel at one unit of time. A car has instance variables to keep track of its behaviour, which are its current X and Y coordinates, width, height, speed, and direction. A car’s direction refers to the direction it is currently headed towards. The first situation is a car that is coming from the West can either head towards East or turn South at the junction. The second scenario is a car coming from the East can either head towards West or turn South at the junction as well. The third scenario is a car can come from the South and either turn left at the junction heading towards West, or keep going straight and then turn towards the East or South again. In summary, cars can only come from East, West and South. When the car changes direction, its direction is updated.

The traffic lights will cycle through 3 lights – red, yellow and green. If the cars at one point reaches a certain position and detects a red light, it will stop until the light turns green. If the light is yellow, the speed is changed to one quarter, simulating the car ‘slowing down’. There are 3 traffic lights, each for one direction of traffic. The lights are tracked via their Boolean values. The combination of red and green lights will last for approximately 6 seconds, while a combination of red and yellow lights will have 3 seconds.

Pedestrians have their own class, and have instance variables coordinate x and y, the width and height and their speed. There is one pedestrian lane, where the pedestrians can walk from East to West and vice versa. If a vehicle reaches a point where there are pedestrians present, it will stop in its current position until there are no more pedestrians on the lane.

# USER FUNCTIONALITIES

The user can add cars and pedestrians using the buttons to the left, which are ‘Add Kancil’, ‘Add Four Wheel’ and ‘Add Pedestrian’.

The user can change the duration of the traffic light changing ‘states’, which are red to yellow, and then yellow to green and back to red. The default waiting value of a traffic light during green and red lights are 6 seconds, and for yellow and red lights are 3 seconds. Thus, if a user sets the new delay timing as 2 seconds, the yellow and red lights will be for 2 seconds, and the red and green lights for 4 seconds, as this program doubles the waiting time from yellow and red lights.

The user can start and stop the simulation with the ‘Start’ and ‘Stop’ button respectively.

# STATISTICS AND ANALYSIS

For statistics, the total number was cars and pedestrians are recorded, according to the number present in the frame. The traffic flow is calculated with the following formula, , where n is the number of vehicles passing by, and t is the time frame. In this program, the time frame is seconds, and the will be converted to hour, thus it is the number of vehicles per hour. For example, if 5 cars pass by in 5 seconds, that equates to 3600 vehicles per hour.

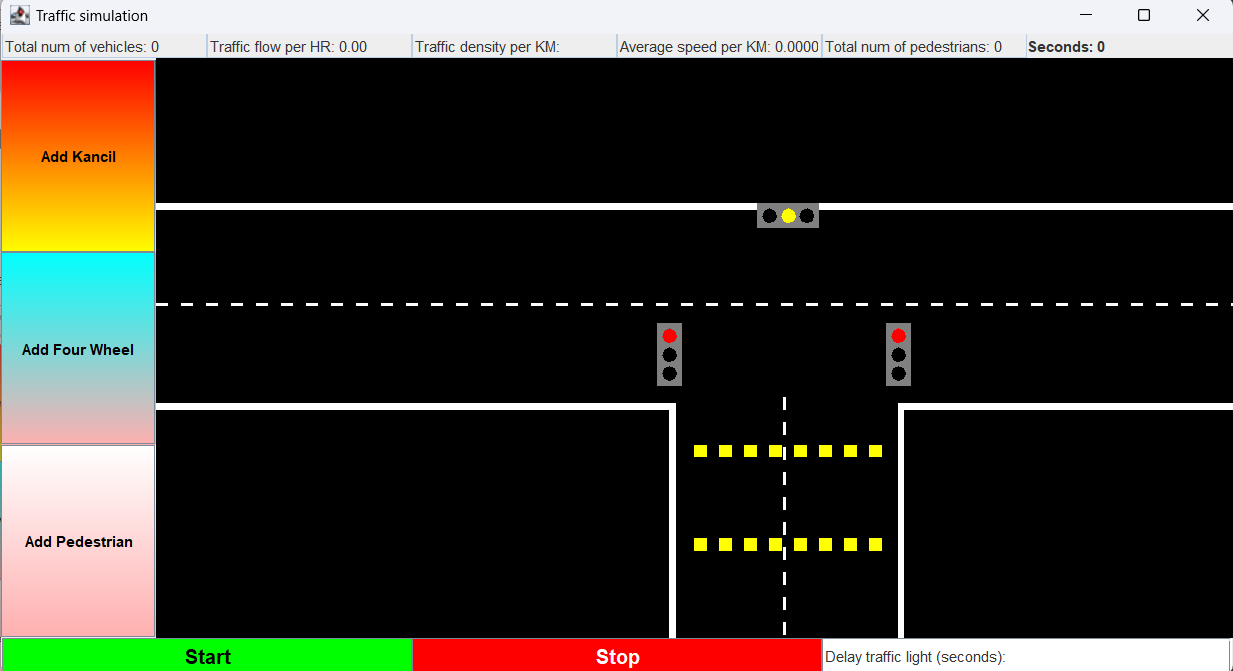
Next, the traffic density is calculated by the number of vehicles that are on a specific road segment. In this program, a pixel is treated as 1 millimetre. The formula for traffic density is , where m is the number of vehicles and L is the road length. For example, if there are 5 cars on a road length of 1000, then there are 5000 vehicles per kilometre.

Finally, the average speed is calculated using traffic flow and traffic density, where the formula is . Thus, if our traffic flow is 3600 vehicles per hour, and our traffic density is 5000 vehicles per kilometre, then we have an average travel speed of 0.72 kilometres per hour.

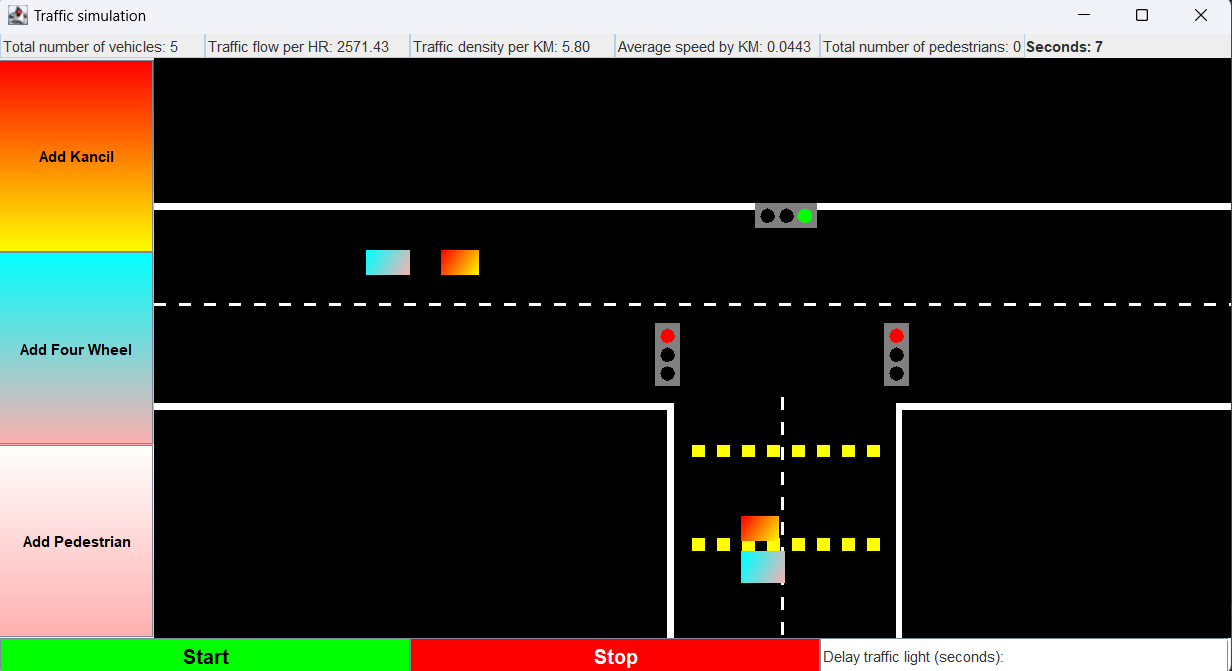
These formulas are referenced from a calculation website (Szyk, 2024).

# SCREENSHOTS

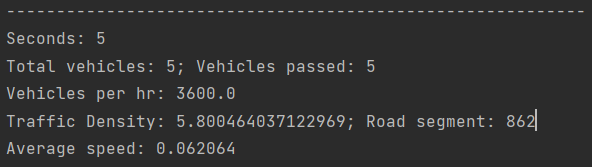
Example screenshot of non-running simulation:



Example screenshot of running simulation:



Example of output from terminal, confirming the correctness of traffic formulas:



# LABELLED TRANSITION SYSTEM (LTS)

A white board with blue writing

Description automatically generated

# REFERENCES

Szyk, B. (2024, January 18). *Traffic Density Calculator*. https://www.omnicalculator.com/everyday-life/traffic-density