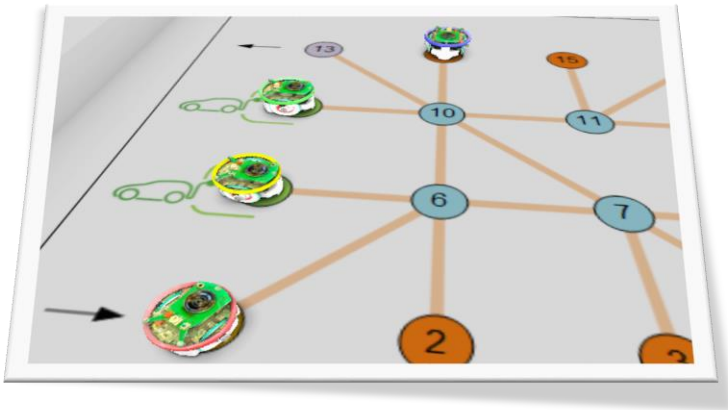


Application of SCOOP to Mission Control in Robotics



Computer Science Research Project

By: Ganesh Ramanathan

Supervised by: Benjamin Morandi

Dr. Sebastian Nanz

Dr. Stéphane Magnenat

Prof. Dr. Bertrand Meyer

Abstract

Concurrent programming brings functional and performance benefits to not just software applications, but also to control of actions in robots. But, as with traditional software applications, concurrent programming carries along the well-known problems like race condition, starvation and deadlock. When such pitfalls manifest themselves in robotic control, it can result in grave physical damage. Consequently, there has been a heightened interest in the robotics community to seek alternatives in safe implementation of concurrency in control. This interest has also gathered momentum due to availability of multiple processors for computation, even on small embedded hardware platforms.

In this research project, we explore the application of Simple Concurrent Object-Oriented Programming (SCOOP) in solving the problem of coordinated and concurrent navigation of several autonomous robotic cars in a parking space. The cars need to execute concurrent navigation actions while utilizing shared resources. The cars act autonomously but coordinate with an entity having an orchestrating role known as the parking manager. While executing its autonomous action, a car can face situation demanding exception handling – for example, handling an obstruction. Though the key concurrency aspect of such problems in robotics has parallels in traditional programming, there are often nuances which emerge when dealing with tangible objects in robotics. For example, noise in measured value, latencies in control-feedback loop, or handling of exceptions etc. brings out the practical requirements. In this project we have used a realistic simulator to test the implementation of a control program written using Eiffel SCOOP and examine its benefits and shortcomings. Studying such real-life robotics programming requirements is essential to the process of fine tuning and understanding the application of any concurrency paradigm such as SCOOP.

Contents

1. Preface - The V-Charge project and the motivation to apply SCOOP	5
1.1 About the V-Charge project	5
1.2 Selecting the concurrency mechanism – why SCOOP?	5
2. An informal introduction to SCOOP	6
3. Problem statement	7
3.1 Goal of this research	7
3.2 Use case for implementation	8
4. Analysing the need for locking during navigation	8
4.1 Empirical analysis	8
4.2 Evaluating Coffmann’s conditions.....	10
4.3 Using a global semaphore	11
4.4 Using Coffmann’s criteria for locking sequence.....	11
4.5 When do we release the lock on a vertex?	12
4.6 The question of precedence.....	12
5. Implementation.....	13
5.1 Object model.....	13
5.2 Handling the vertices as global resources.....	14
5.3 Locking and navigation.....	15
5.3.1 Variant-1: Set a flag on the vertex and use it as pre-condition	15
5.3.2 Variant-2: Start navigation only after locking all vertices	16
5.3.3 Variant-3: Navigation using a single globally visible flag.....	17
5.3.4 Variant-4: Navigation without the use of synchronization flags.....	18
5.4 Handling obstruction.....	18
5.5 Control law	19
5.6 Environment.....	19
5.6.1 Communication protocol	20
5.7 Interface to Eiffel program.....	21
6. Comparison to traditional multi-threaded approach.....	21
6.1 Navigation	21
6.1.1 Safety.....	22
6.1.2 Wait conditions	23
6.1.3 Asynchronous actions	23
6.2 Handling obstruction.....	23
6.3 Control loop.....	24

7.	Evaluating the application of SCOOP to mission control scenarios	25
7.1	Messaging between entities	26
7.2	Safety.....	26
7.3	Deadlocks	26
7.4	Exception handling.....	26
7.5	Summary	26
8.	Conclusion	27
9.	References.....	28
10.	Appendix-A: Setting up the simulation environment	29
11.	Appendix-B: Setting up the Eiffel project “scoop_park”	31

1. Preface - The V-Charge project and the motivation to apply SCOOP

1.1 About the V-Charge project

V-Charge [1] is a collaborative project which seeks to allow easy access to electric cars to foster environment friendly mobility. In order to ease the process of hiring an electric car and depositing it back after use, the project envisages creation of dedicated parking spaces (for example, in the proximity of a railway station or an airport) where the user can request for a car, obtain one, drive around, and finally return it back to the same or another parking lot. The process of hiring and returning the car is automated to the maximum extent. On user request, the car autonomously drives itself from its parking space to a place where the user takes over. After using the car, the user can return it by simply bringing it to a drop-off point in the parking space, after which the car once again autonomously drives itself to a temporary parking spot or to a recharge station. Easing the process of picking up a car, dropping it back or docking it to a recharge station is expected to motivate people to adopt the shared e-car as an option for mobility. As a proof-of-concept, an operational system is targeted to be installed on the campus of ETH Zürich and TU Braunschweig.

The project has several dimensions associated with it – such as designing and implementing low cost sensory system, autonomous navigation system using maps, and global planning mechanism. Within the parking space, it is foreseen to combine on-board navigation sensor (like GPS) along with a camera watching over the space itself. The planning system needs to be adaptive and also needs to work in synchronization with local planning on-board the car. For example, the car should be able to handle dynamic obstacles like other cars or pedestrians on its path while following instructions from the parking manager to serve user requests.

In this project scenario we see an exciting possibility to study the use of a truly object-oriented concurrency mechanism to address issues like parallel execution of actions, protected access to shared resources and coordination between system components. For this purpose we implemented a control program using SCOOP [2][3] and tested it on a realistically simulated environment.



Figure 1: Illustration of the parking space for the V-Charge project (from [1]).

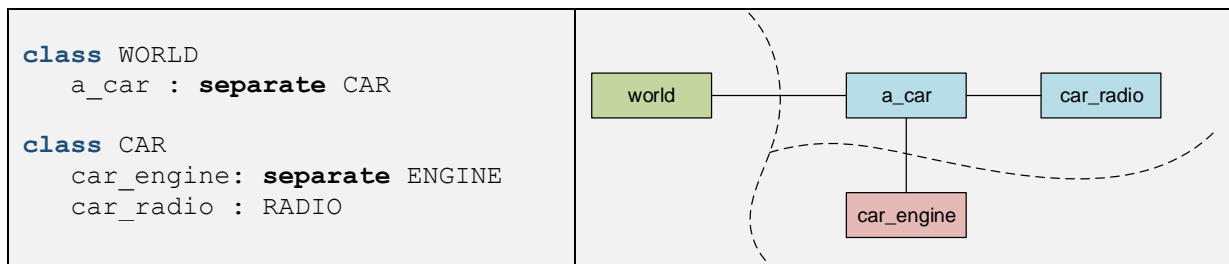
1.2 Selecting the concurrency mechanism – why SCOOP?

The impedance mismatch between object-oriented modelling and concurrency is well known in the software development community. While object-oriented modelling enables representation of real-life objects and their inter-relationships, it does not cover the description of concurrent actions that may happen in these interactions. One such study [3] examines concurrent programming using Java™ in

comparison to SCOOP. For example, in this study it was noticed that SCOOP’s integration of synchronization condition in the language avoided the mistake of inadvertently omitting *wait* or *notify* which occurred frequently in Java’s multi-threaded approach. We have in the past conducted several experiments using SCOOP for solving concurrency problems in robotics [4][7][11]. These experiments have shown definitive advantages in using SCOOP for construction of software for robotics control. Primarily, adopting SCOOP as the concurrency paradigm allows the object model to remain the central aspect of the software design. Routine calls on objects designated as “separate” not only results in asynchronous execution but occurs only after inherent locking of the target object’s processor. In addition to this, the contractual pre-condition of the called routine acts as a wait condition when it involves separate objects. These mechanisms result in guaranteed avoidance of race conditions and allow control synchronization without the use of semaphores (thus reducing code complexity). In comparison, adopting the long practised approach of multi-threading shifts the focus away from the object model to the thread execution model. In view of these proven advantages, we wish to solve the problem of autonomous navigation of cars using SCOOP. This exercise will also allow us to further study advantages or shortcomings of using SCOOP in robotics control programming.

2. An informal introduction to SCOOP

The core idea behind SCOOP revolves around the concept of objects being handled by a processor. The processor is any mechanism that can execute instructions sequentially – to this end, a processor may be a thread, a process or a physical processor itself. In most object-oriented programming language, an object instance can instantiate or create another object and refer to it using a member variable. In SCOOP, if the instantiated object is to be handled by a different processor, then the relationship is additionally qualified as being “**separate**”. In the following example, the instances of CAR and RADIO are handled by the same processor whereas the instances of WORLD and ENGINE are handled by distinct processors.



In the above example, the processor boundaries are indicated with dotted lines. Any relationship which crosses processor boundary is qualified as being **separate**. For example, `a_car` and `car_engine` are **separate** with respect to each other whereas `a_car` and `car_radio` are non-separate in relationship. In this sense, the `car_engine` is also **separate** with respect to the `world`. In order to simplify the description of the specialities in the interaction with a **separate** object, we will use the following example code snippet. Note that the term “Command” is used for routine calls on an object that can have side-effects and may or may not return a result, whereas the term “Query” is used to indicate a routine call that returns a result but does not cause any side-effects.

```

class WORLD
  initialize_car(c : separate CAR) -- The separate object has to be a parameter.
  require car_may_initialize : c.is_ready -- Will wait because c is separate.
  do
    c.check_systems() -- Will be executed asynchronously at some point of time.
    ...
  has_started := c.start() -- Command with result. Will be executed synchronously.
  car state := c.system status -- Query. Will be executed synchronously.

```

Interactions with a **separate** object have the following characteristics:

1. Command calls invoked on a **separate** object, but not involving any parameters of reference type and without any return of result, are asynchronously executed.
2. Queries on **separate** objects are always synchronously executed.
3. In order to access a **separate** object, it has to be a parameter of the enclosing routine. Such a routine would wait until it can obtain lock on the processor of each **separate** parameter and then hold the locks until the completion of the routine.
4. Pre-conditions in the Eiffel programming language serve to assert requirements that a caller needs to satisfy before the called routine can be executed. But, if the target of the pre-condition evaluation is a **separate** object, then instead of throwing an exception, the execution waits for the predicate to be true. During this wait, the lock on the **separate** object is not held.

The concept of an object instance having a handler, together with the above stated behaviour, allows us to implement a concurrent program which is free of race-conditions (because of guaranteed mutual exclusion) and also has a well-defined manner of accessing shared resources. A complete discussion of the mechanisms in SCOOP can be found in [2], [3] and [13].

3. Problem statement

The autonomous parking system for electric cars consists of a parking space with designated spaces for temporary parking and re-charge stations. In addition, there are specific places where a user can pick-up or drop-off a car. An electric car is brought to the drop-off space by the user. After this, the car needs to move autonomously to either a temporary parking place or to a recharge station (Figure 2 below shows an example parking space). This decision is made by the parking manager which is a software program orchestrating all activities in the parking space. Similarly a user can request a car to be brought to the pick-up area, upon which the parking manager will instruct a specific car to navigate itself to the pick-up area. In this system more than one car might be required to navigate autonomously in a concurrent fashion.

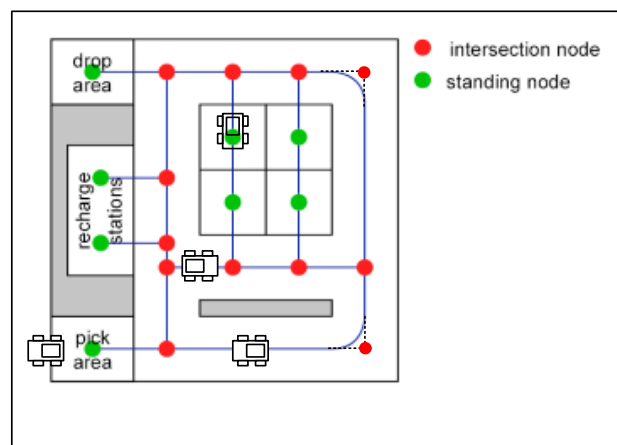


Figure 2: Example layout of parking space (from Dr. Stéphane Magnenat, ASL ETHZ)

3.1 Goal of this research

Our goal is to design and implement the control software to enable electric cars to autonomously navigate in the parking space according to instructions from the parking manager. This has to be achieved by taking care of concurrency, coordination and safety issues. In specific, we examine the use of SCOOP in addressing the problem of concurrent navigation of the cars and its coordination with the

parking manager and in general we look at the requirements of mission control of autonomous vehicles. We also compare the implementation in SCOOP with an equivalent program written in Microsoft .Net™ using a multi-threaded approach. The experience derived from this exercise enables us to evaluate the advantages and challenges of using SCOOP to solve problems of concurrency in robotics.

3.2 Use case for implementation

In order to present a concrete use case for the software program, a user story is described here to highlight all coordination and concurrency issues.

A car is brought to the drop-off node by the user (using the joystick control). The parking manager senses the car on this node and based on its battery charge status and availability of recharge stations, instructs the car to either proceed to a charging node or to a temporary parking node. The car then autonomously navigates to the target node. When the car is in a charging node, it periodically increments its own charge value and then on reaching a threshold informs the parking manager. Once the car is recharged, the parking manager takes note of this message and instructs the car to move to a temporary parking node or to the pick-up node. A car is sent to the pick-up node if there is a user request for the car. When a car comes to the pick-up node, it relinquishes autonomous control and allows the user to manipulate it (again, with the joystick).

During autonomous navigation, each car will plan its path (it receives just the target node) and then navigates through it after "securing" nodes on the path. The process of securing (or locking) ensures that no other car will attempt to utilize that node concurrently. On traversing a node, the car "releases" the lock on the node so that it is free to be used by other cars. While attempting to lock an already locked node on its planned path, the car will wait indefinitely until it can obtain the lock.

On encountering an obstacle in its path, a car will stop immediately and wait for the obstacle to be cleared. It shall be possible to terminate the system even when a car is waiting for an obstacle to be cleared. In case an obstacle is not resolved within a pre-defined time period, the entire system will be shut down after all cars currently in motion have reached their target node or to an intermediate node.

4. Analysing the need for locking during navigation

At any given time, it is possible that more than one car needs to execute autonomous navigation. For example, a car may be required to navigate from the drop-off location to a recharge station while another car might be required to move from a temporary parking space to the pick-up location. In the most defensive manner, we could execute these two actions sequentially – i.e. the first car completes its navigation and then the next car begins its sequence of actions. But in face of high demand for movement of cars, this approach will result in increased delays. Hence, we should consider parallel autonomous navigation of multiple cars. The primary requirement of concurrency applies to this scenario also - we need to ensure safety such that two or more cars may never access a vertex or an edge in the path simultaneously. While doing this they need to avoid deadlocks and maintain fairness amongst contenders.

4.1 Empirical analysis

In the following discussions, we will use a simplified path graph to illustrate different scenarios where concurrent navigation of two cars are involved. We first begin by looking at scenarios empirically - this will help us examine the requirement for locking, following which we will attempt to formulate the conditions formally. Note that an intersecting vertex (e.g. vertices 2, 3 and 5 in Figure 3), cannot be the target of a navigation (i.e. a car uses such vertices only to traverse to the next connected vertex). Also, the parking manager will not instruct a car to move to target vertex which is already occupied.

In the simplest case, illustrated in Figure 3, two cars need to traverse vertices and edges completely disjoint from the other.

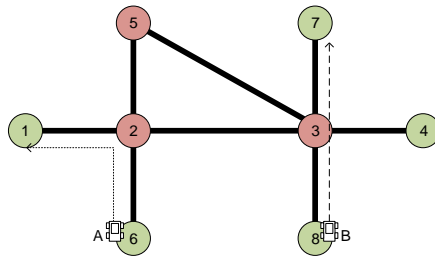


Figure 3: Concurrent navigation without need for locking

In this case the cars can navigate without having to wait for the other, and hence no locking or coordination is required. But, it could happen that two cars share one or more vertices in their respective paths. In this case each of the cars has to wait until the next shared node its has been freed.

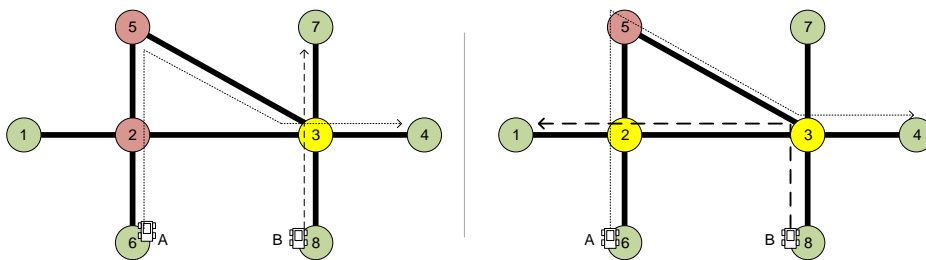


Figure 4: Concurrent navigation with need for locking on one (left) or more vertices (right)

We thus see empirically that it is not possible to arrive at deadlock when two cars share only vertices on their path and it seems that is sufficient if the cars only lock the next node on their path. In the following diagram we can see that even when the cars have to use a common edge in the same direction, they need only lock the next vertex. But this assumption fails when the cars share an edge that they have to use in the opposite direction.

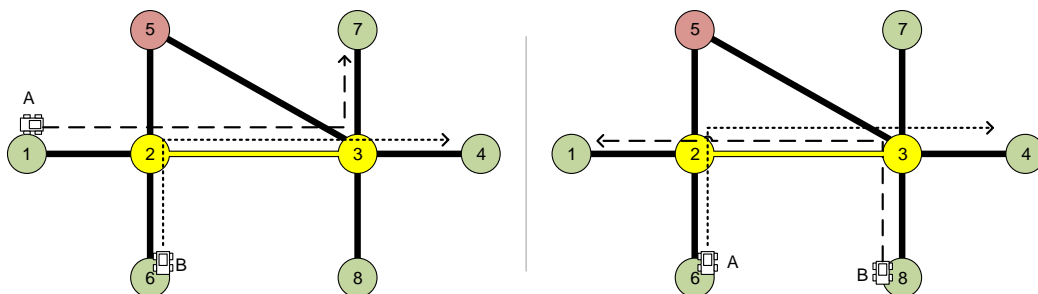


Figure 5: Concurrent navigation involving shared edge traversal - in same (left) and opposite (right) direction

However, if we follow the strategy of obtaining lock on the next vertex on the path then we can quickly arrive at deadlock when in a circular wait condition. This can happen either when two cars need to share an edge travelling in opposite directions or when more than two cars need to wait on one another. This scenario is illustrated in Figure 6.

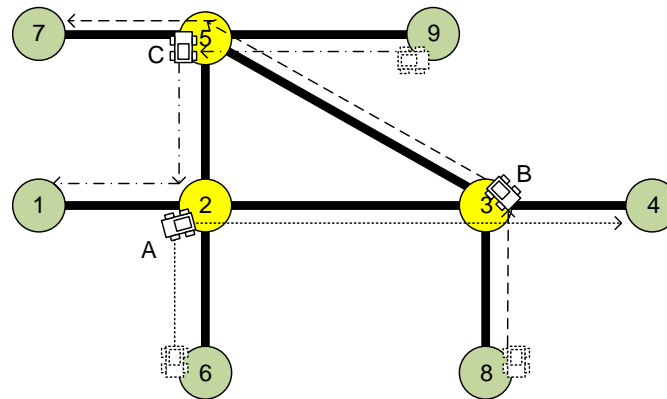


Figure 6: Deadlock scenario with circular wait conditions

Hence the empirical assumption that we need to lock only the next vertex is proven false. We can now propose that each car should lock all vertices on its path before it commences navigation. But, since the locks on the vertices have to be obtained sequentially, the process of locking itself would be fraught with risk of deadlock. To illustrate this, let us consider the scenario illustrated in Figure 6. The path for Car “A” is the ordered set of vertices {2,3,4}, for “B” it is {3,5,7} and finally for “C” it is {5,2,1}. So, if we let all three cars obtain locks concurrently, there is bound to be a deadlock when each car has obtained lock on the first vertex in its path and waits for the next.

4.2 Evaluating Coffmann’s conditions

Moving away from empirical evaluation, we now examine the relevance of Coffmann’s “Necessary and sufficient” conditions for deadlock [5] with regard to the above scenario.

1. *Mutual exclusion: A vertex or an edge can be occupied by only one car (and edge if they are moving in opposite direction).*
2. *Hold and wait condition: The car holds the lock for the node it is currently in and is waiting for the lock on the next node.*
3. *No pre-emptive condition: A car cannot be removed from a vertex.*
4. *Circular wait condition: The car is waiting for lock on a vertex and this vertex is held by another car which is waiting for the vertex held by first car.*

In other words, if occurrence of at least one of the above four conditions can be avoided, we can guarantee a deadlock-free system. From the knowledge of our system we know that condition of mutual exclusion cannot be avoided as each vertex can physically be occupied by only one car. Also, we know that once a car occupies a vertex, it cannot be pre-empted (or removed temporarily) out of the vertex.

This leaves us with two possibilities – either avoid the hold-and-wait condition or avoid a circular wait condition.

Avoiding hold-and-wait condition is offered by SCOOP when dealing with routine calls with separate parameters. The routine call will wait until locks on all such separate parameters are obtained, but will not hold them during the wait process. The routine progresses only when locks on all the separate parameters are obtained. Having said that, there are two hurdles when trying to utilize this mechanism:

1. We do not have fixed set of parameters, but a varying number of separate vertices (depending on the path). There is no possibility in SCOOP to handle locking on elements of a collection passed as parameter.
2. Once a vertex has been traversed, we wish to release the lock on it (not an essential requirement, but would be favourable). In case of the routine call with separate parameters, the lock on all the parameters is held until the completion of the routine.

Hence we need to work-around this restriction to ensure that we do not hold the lock on vertex while waiting for lock on another. We will do this by making sure that we start obtaining locks only when we are sure that we do not have to wait.

To avoid a possible circular wait condition, we can adopt two approaches – either we allow only one car at a time to obtain all its required locks (using a global semaphore), or order the sequence in which locks are obtained (Coffmann’s proposal [5]). We discuss these two strategies further in the following sections.

4.3 Using a global semaphore

This approach can be explained with the example shown in Figure 6. Suppose that car B is the first to get through the global semaphore and obtains all its locks (on vertices 3, 5 and 7). The next car, say C, which is waiting on the semaphore then starts to obtain locks (on vertices 5, 2, and 1) but has to wait till B relinquishes its lock vertex 5. The progress of C is then determined by the question as to when B will release its lock on 5. The progress of A is consequently dependent on C – it has to wait till C releases the global semaphore.

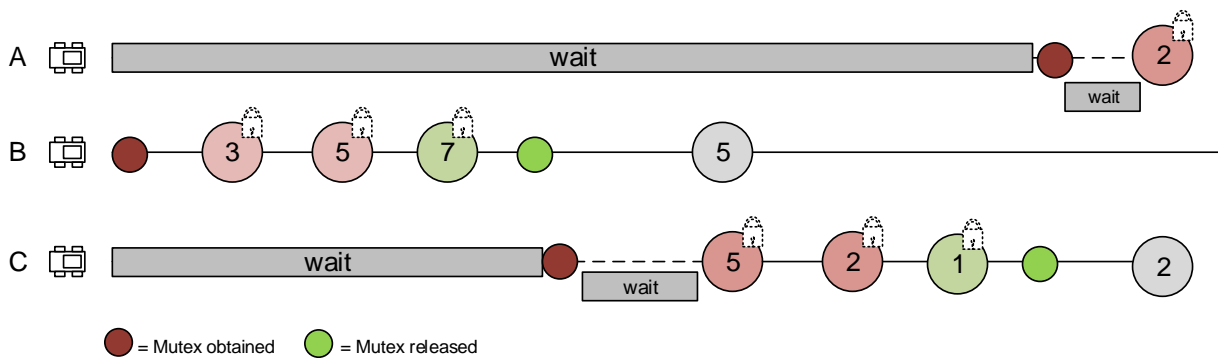


Figure 7: Using global mutex to avoid deadlock

This strategy serves the purpose but has two disadvantages:

1. In case a car has no vertices in its path that is common with others, it might have to wait unnecessarily.
2. Use of semaphore is incurred and consequently makes the program dependent on its correct usage.

4.4 Using Coffmann’s criteria for locking sequence

Continuing with the previous example, if we re-order the vertices according to increasing order of their identifiers then each car would have to lock the vertices in the following sequences:

A = {2, 3, 4}, B = {3, 5, 7} and C = {1, 2, 5}

In this case all the three cars are allowed to go ahead and compete for locks, and depending on the scheduling of executions, one of them succeeds in locking all its required vertices without any circular deadlock. In case one of the cars does not have any vertex common with others it will obtain all its locks without waiting, thus eliminating the disadvantage faced by the global semaphore method described in section above.

4.5 When do we release the lock on a vertex?

Next, we need to consider what happens to an obtained lock once the vertex has been traversed – i.e. once the car leaves the vertex on which it had a lock, should it keep the lock until it completes its navigation of all vertices, or should it release it once it has traversed the vertex? We presume that a vertex appears only once in the navigation path (i.e. the vertex need not be traversed more than once).

Since locks on all vertices on the path are acquired upfront, releasing the lock on a traversed node will not endanger safety or risk deadlock. And in general sense, releasing the lock on resource after it is no longer needed does not pose any risk to safety. In fact, this can lead to improved performance as another car waiting to obtain lock on a vertex progresses faster. However, a vertex is not a purely logical resource – a car physically releases the use of a vertex when it is out of certain geometric bounds of the vertex. To explain this, consider the following scenario (illustrated in Figure 8) where car A releases the lock on vertex 2 as soon as it starts its progress towards 3. Now, for some unexpected reason car A cannot move forward (say, due to a failure) but car B has grabbed the lock on 2 and starts moving towards it. Eventually B will stop (sensing obstruction). But now, B has released its lock on its originating vertex. This can lead to a dead-lock if the paths are closed (i.e. there will be cars waiting on the edges)

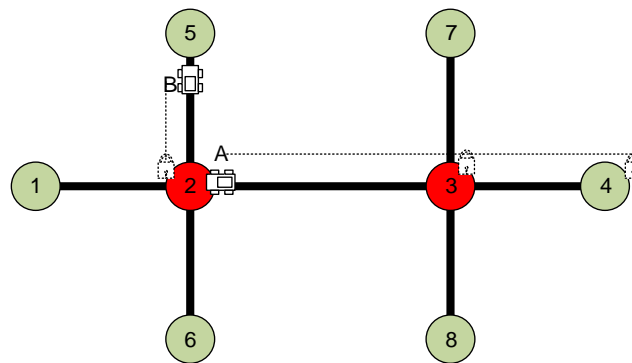


Figure 8: Scenario where car "A" releases lock on vertex 2, but does not leave it due to failure.

We can defensively handle scenario by informing the parking manager about inability to progress on the path. The parking manager then sets a flag which then prevents all cars from leaving their current vertex.

A more elegant solution would be when the car updates the vertex (the currently occupied one and the next approaching one) about its pose. The vertex can then determine its state of being occupied (which is used as a pre-condition before another car attempts to use the vertex).

4.6 The question of precedence

Adopting Coffmann's criteria for the sequence in which locks are obtained will work as long as none of the contenders occupy a resource desired by another while also contending for one or more other common resources. The following scenario illustrated in Figure 9 explains the problem:

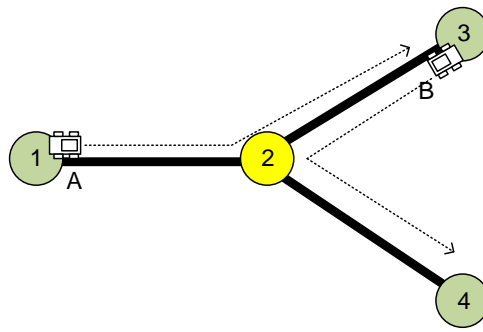


Figure 9: Possible deadlock scenario

Car A wishes to navigate to vertex 3 which is being currently held by B, while car B wishes to navigate to vertex 4. Here, if precedence for B is not ensured, it will lead to a dead-lock. One could simplify the problem by making it imperative that the target vertex should not be already held by another car. Alternatively, we can use the wait on pre-condition concept in SCOOP to wait on the target vertex to be flagged as available. This solution is described in the variant-2 of the implementation in the following section. Note that this problem too can be solved if SCOOP provides a way to obtain locks on elements of a collection passed as parameter.

5. Implementation

5.1 Object model

The entity types and their inter-relationship are shown in UML diagram in Figure 10. The central entity is the single instance of PARKING_MANAGER together with multiple instances of CAR owned by it. The PARKING_MANAGER and each instance of the CAR have a local MAP representing the path in the parking space as a graph. Since each instance of VERTEX needs to be treated as a global resource (protected from concurrent access), the PARKING_MANAGER instantiates a collection of separate objects representing the vertices.

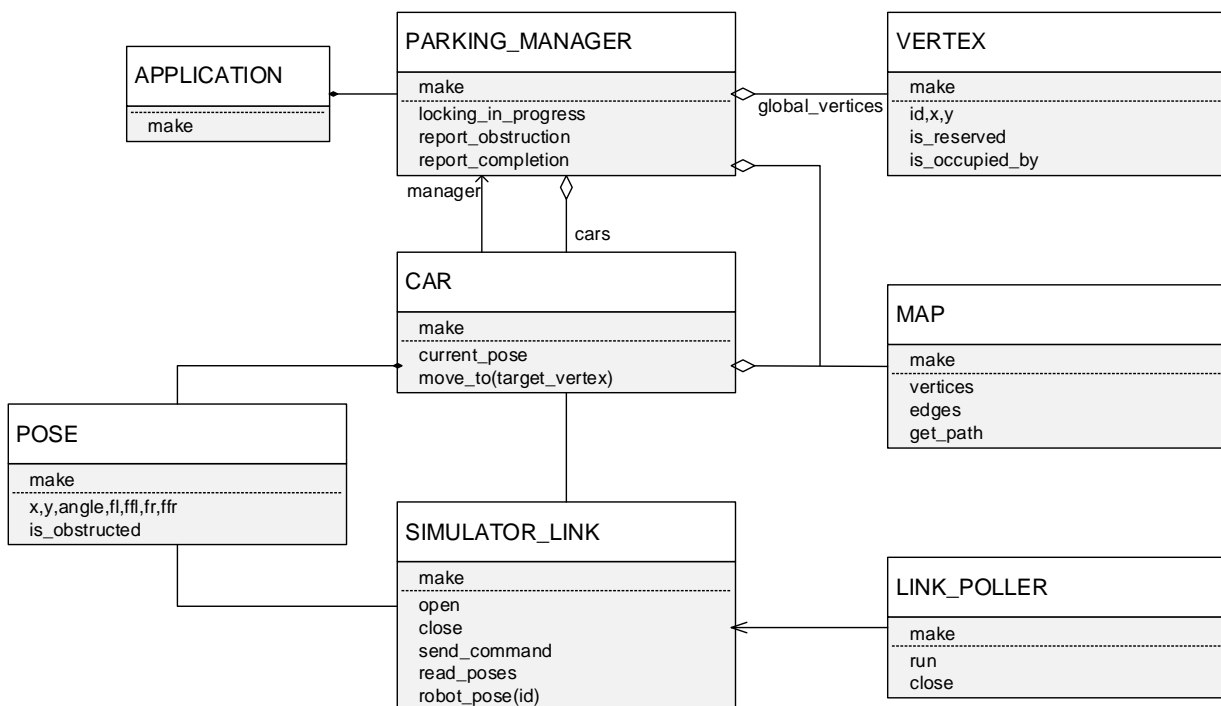


Figure 10: The object design showing essential classes

The APPLICATION represents the entry point for the program. It also serves to take in instructions from the user for the purpose of testing the system. An instance of PARKING_MANAGER is created by the APPLICATION. The APPLICATION also creates an instance of the SIMULATOR_LINK and an associated LINK_POLLER (to constantly read the pose updates). Each CAR is giving the reference to the SIMULATOR_LINK which it then uses to obtain its POSE. The pose, in strict robotics terminology, is a tuple of the robot’s position (in Cartesian coordinates) and its angular orientation, but we have extended this to include obstruction distance as measured from the four sensors.

The proposed object model above is derived by studying the real-world entities. Until now, we have not considered program execution and concurrency of actions in the object model (except for recognizing that vertices are resources that need to be protected from concurrent access). We wish to highlight the fact that this model remains unchanged in the next steps when overlaying it with the design of concurrent actions using SCOOP.

5.2 Handling the vertices as global resources

Each instance of the VERTEX is a separate object whose reference is stored in a list (which is instantiated and held by the PARKING_MANAGER). Figure 11 shows the instance of parking manager along with multiple instances of cars and vertices. Objects handled by the same processor are shown within dotted boundaries. In entire system there is one processor (p) handling the single instance of parking manager, whereas each instance of car and vertex is handled by its own dedicated processor (q₁..q_n and r₁..r_n).

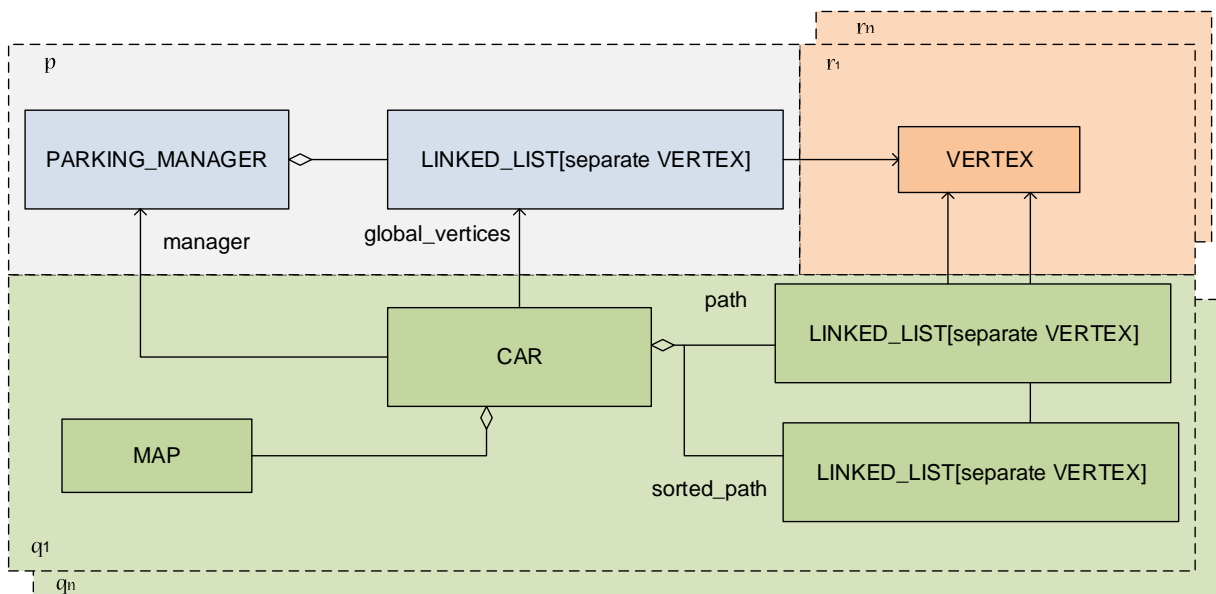


Figure 11: Inter-relation between object instances. Shading indicates objects in the same processor boundary.

The CAR uses its instance of MAP to find the ids of the vertices which are on the shortest path to its target vertex. It then builds two lists containing references to the global vertex objects – one of the list contains the reference in the order of navigation path while the other contains the references in the order of vertex ids. As an example, let us suppose CAR instance A wishes to navigate from its current vertex 6 to target vertex 7 (see Figure 12).

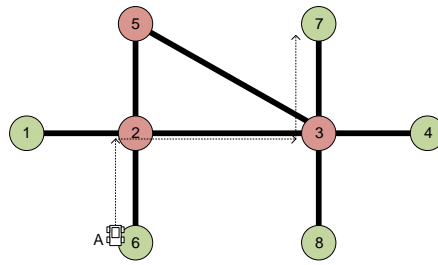


Figure 12: Example of navigation path

The routine call on the MAP returns the shortest path as a collection of ids – in this case {6,2,3,7}. Following this, the list *path* is populated with references to vertices whose ids are 6, 2, 3 and 7 (in that order) and the list *sorted_path* is populated with references to separate vertices whose ids are 2, 3, 6 and 7.

5.3 Locking and navigation

While analysing the possible implementations to lock the vertices prior to navigation, several variants were considered. These variants are described below to provide a glimpse of the design thought process when using SCOOP. We start with the implementation that evolved out of the first intuition and then go on to refine it in further variants.

5.3.1 Variant-1: Set a flag on the vertex and use it as pre-condition

The core idea of this approach is to use a flag on the vertex to indicate if it has been “reserved” by a car for purpose of navigation. A car wishing to use the vertex has to wait for the flag to be set to false. The following code snippet from the implementation highlights the key aspect of this approach:

```

--Routine to cause the car to navigate autonomously to the target node.
move_to_target(target_vertex_id : INTEGER)
do
    retrieve_path_from_map(start_vertex_id, target_vertex_id)
    --Set flag on the vertex in sorted order and then navigate according to
    --path order
    from j := 1
    until j > sorted_path.count
    loop
        reserve_vertex(sorted_path.i_th (j))
        j := j + 1
    end

    from j := 1
    until j > path.count
    loop
        move_to_vertex(path.i_th (j))
        j := j + 1
    end
end

reserve_vertex(sv : separate VERTEX)
require
    vertex_not_used : not sv.in_reserved --Execution will wait if required
do
    sv.is_reserved := true;
end

move_to_vertex(sep_v:separate VERTEX)
do
    move_to_xy(sep_v.x, sep_v.y)
    sep_v.is_reserved:= false
end

```


The VERTEX has a Boolean flag called `is_reserved` to indicate if the vertex is intended for use by a CAR. The key part of this approach lies in the pre-condition of the routine `reserve_vertex`. In this routine we use the `is_reserved` flag in pre-condition evaluation. Since the subject of this evaluation is uncontrolled (is separate), the pre-condition behaves as a wait condition. In case the vertex is in use, the pre-condition check will wait until the flag is set to false (by the car which had reserved it in the first place). Having set the flag on all vertices in the path (in the order of their ids), we are assured that other cars wishing to use one or more vertices in common will have to wait.

We see two important features of SCOOP in action here – 1. The pre-condition functions as wait condition, which acts as a synchronization mechanism without the need for semaphore like signalling, and 2. The guaranteed absence of possible race condition in setting the flag which is brought about by the need to lock the target processor before the assignment is carried through.

Taking a step back, we see that in order to implement the concurrency and coordination requirement we have not modified the object model originally laid out. The inclusion of a member variable `is_reserved` to indicate the state of the vertex is also consistent with notion in the domain (i.e. the vertex being a physical part of the path, has in reality the states of either being used by a car or free). There is however a lack of contractual binding on the consumers of a vertex to set the flag `is_reserved` before they proceed to access the vertex. By setting the flags on a series of vertices, an instance of car has only indicated its desire to use those vertices. Clearly, a caller which does not follow the rule (of first setting the flag) and proceeds to call the routine `move_to_vertex` can get the lock on the vertex. Notice that despite this loophole, SCOOP ensures safety – the vertex will never be approached by two or more cars because in order to move to vertex a car needs to hold the lock on its processor (and only one car can own the lock on the processor).

5.3.2 Variant-2: Start navigation only after locking all vertices

Since we cannot not trust that all users of the vertex will respect the `is_reserved` flag, we need to make sure that we indeed have exclusive access to all vertices before the car starts to navigate. Fortunately, a useful rule in SCOOP comes to play [6]:

 **Rule -- Wait:** A routine call with separate arguments will execute when all corresponding processors are available and hold them exclusively for the duration of the routine.

(From <http://docs.eiffel.com/book/solutions/concurrent-eiffel-scoop>)

In our scenario, we need to obtain lock on the collection of vertices in the path. As mentioned before, there is no mechanism in SCOOP by which individual elements of the collection passed as a parameter are considered for locking. For example if a parameter is of type `LINKED_LIST[separate VERTEX]` then the vertices are not considered for locking. This would be a desirable feature in SCOOP which will allow treatment of variable number of separate parameters. We can partly work-around this by using recursive function call as show in the code snippet below:

```
move_to_target(target_vertex_id : INTEGER)
do
  from j := 1
  until j > path.count
  loop
    reserve_vertex(sorted_path.i_th (j))
    j := j + 1
  end

  lock_all_navigate(path.count, path[path.count])
end
```



```

lock_all_navigate(n:INTEGER; p:separate VERTEX)
do
  if n > 1 then
    lock_all_navigate(n-1, path.i_th (n-1))
  end
  move_to_xy(p.x,p.y)
end

```

Compared to variant-1, what we achieve here is the guarantee that a vertex on the path is not accessed by caller that does not follow the design rule of first setting the flag on the vertex.

We saw earlier that despite using Coffmann's criteria for sequence of locking, we can still end up with deadlock if the target vertex is occupied by another car. We can solve this easily by adding another flag on the vertex to indicate if it is occupied and then have callers wait on this flag as a pre-condition.

```

...
  wait_for_target_and_reserve_vertices(path[path.count])
  lock_all_navigate(path.count, path[path.count])

reserve_vertex(sv : separate VERTEX)
  require
    vertex_not_reserved : not sv.is_reserved or sv.occupied_by = id
  do
    sv.is_reserved := true;
  end

wait_for_target_and_reserve_vertices(sep_target : separate VERTEX)
  require
    target_is_free : not sep_target.is_occupied
  local
    j : INTEGER
  do
    from j := 1
    until j > path.count
    loop
      reserve_vertex(sorted_path.i_th (j))
      j := j + 1
    end
  end
end

```

5.3.3 Variant-3: Navigation using a single globally visible flag

In this variant, we will try to avoid setting a flag on each vertex. To do this, we create a flag on the PARKING_MANAGER (which is visible to all instances of CAR). Each instance of car desiring to lock the vertices in the path has to set this flag, if required, by waiting.

```

move_to_target(target_vertex_id : INTEGER)
do
  signal_manager(manager)

  lock_all_navigate(path.count, path[path.count])
end

lock_all_navigate(n:INTEGER; p:separate VERTEX)
do
  if n > 1 then
    lock_all_navigate(n-1, path.i_th (n-1))
  end

  release_manager(manager)

  move_to_xy(p.x,p.y)
end

```

```

signal_manager(parking_manager : separate PARKING_MANAGER)
require
    manager_not_locked : parking_manager.locking_in_progress = false
do
    parking_manager.locking_in_progress := true
end

release_manager(parking_manager : separate PARKING_MANAGER)
do
    parking_manager.locking_in_progress := false
end

```

Compared to variant-2, though we have done away with the need to set a flag on each vertex, we have lost the state information on it (i.e. the information if it is being intended for navigation).

5.3.4 Variant-4: Navigation without the use of synchronization flags

Finally, we consider the variant in which we use a recursive method to obtain locks on vertices in the order of their ids (Coffmann's criteria) and then execute navigation in the order of path ids. The principal disadvantage here is that the locks on vertices are released only after the completion of navigation.

```

move_to_target(target_vertex_id : INTEGER)
do
    lock_sorted_navigate(sorted_path.count, 1)
end

lock_sorted_navigate(n:INTEGER; p:separate VERTEX)
do
    if n < sorted_path_node_ids.count then
        lock_sorted_navigate(n+1, sorted_path.i_th (n+1))
    else
        from j := 1
        until j > path.count
        loop
            move_to_vertex(path.i_th (j))
            j := j + 1
        end
    end
end
end

```

5.4 Handling obstruction

When the path of a car is obstructed, the values obtained from the IR distance sensors indicate the distance to the obstruction. In case the sensed distance is below a certain threshold value, we need to stop the navigation of the car. The obstruction can be temporary – in which case the car waits until the obstruction is cleared and then proceeds with the navigation. On the other hand, if the obstruction is long-standing, we need to handle this as an exception in the system. In our implementation we have worked around the problem by having the car set a flag on the parking manager when it encounters a long-standing obstruction. The parking manager then sets a status flag on itself (indicating that system has to shutdown). This status flag on the parking manager is queried by each car in its navigation routine, and in case the status flag is set to indicate shutdown, the car does not proceed with the navigation to the next vertex on its path.

At this point we experience two particular difficulties:

- The parking manager cannot invoke a routine on the cars (to ask them to stop) if they are busy in their control loop. Here it would have been advantageous to have a possibility for parking manager to call a routine on the car even though it is busy.

- When a car faces long-term obstruction, it has to inform the parking manager about the anomaly by making a routine call. And for this, it might have to wait to obtain the lock on parking manager. Such a wait is not desirable when having to deal with conditions that require immediate control action.

5.5 Control law

The control law is an algorithm used for computing the wheel speeds in a closed loop control. The car in our simulator is a two wheel differential drive robot and its direction of motion is controlled by applying a difference in speed between the two wheels. Figure 13 illustrates the geometry behind the computation.

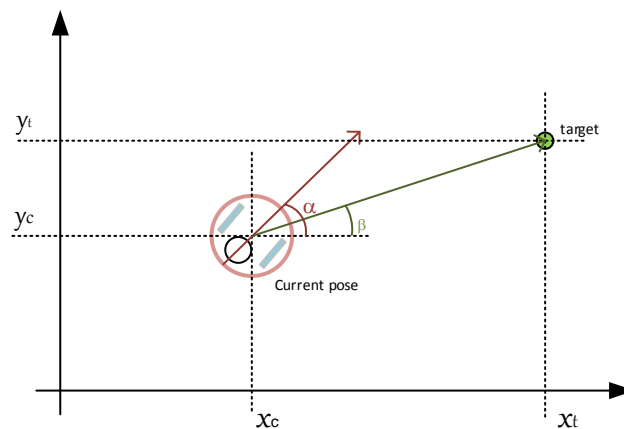


Figure 13: Geometrical consideration for control law

Let $P=[x_c, y_c, \alpha]$ represent the current pose of the car and $T=[x_t, y_t]$ represent the desired target location. The instantaneous values of the wheel speeds required to correct the motion path is given by:

$$\text{Target angle } \beta = \text{atan}((y_t - y_c)/(x_t - x_c))$$

Hence, the correction angle $\delta = \alpha - \beta$. If s is the required speed of travel, then component speeds are:

$$s_x = \cos(\delta) \cdot s$$

$$s_y = \sin(\delta) \cdot s$$

The left and right wheel speeds would then be:

$$S_l = S_x - S_y$$

$$S_r = S_x + S_y$$

Note that the value of the angle obtained in the pose update has to be normalized (to range $0..2\pi$)

5.6 Environment

The actual electric cars and the parking space is simulated using a program based on Enki [18]. The simulation environment consists of a viewer in which the robots can be visualized and can be instructed to move in the pre-defined space. The simulation wrapper allows an external program to send commands to move one or more robots. The external program needs to connect to a TCP socket on the simulator and send the commands as strings. The simulator also sends updates on the TCP socket

connection. The updates contain the robot poses and are sent at periodic intervals (a “pose” is a data tuple consisting of the robots Cartesian position and orientation). The complete instructions to setup the simulation environment is explained in detail in Appendix A. The schematic in Figure 14 shows the overview of the environment – the simulation viewer and the Eiffel (SCOOP) program run in different process spaces which communicate with each other using simple text-based protocol on TCP sockets.

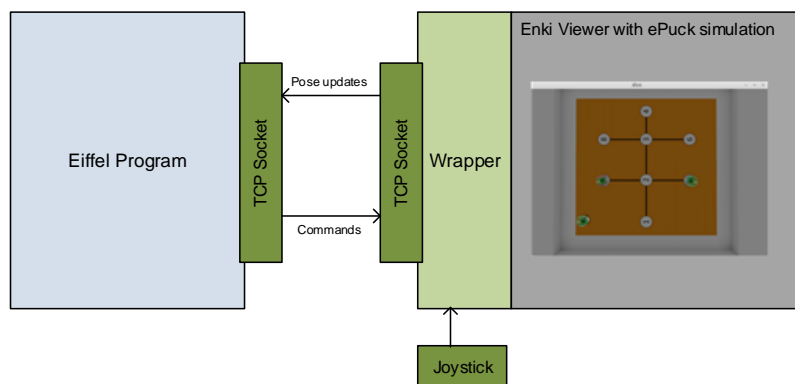


Figure 14: Overview of the simulation setup

The environment can be initialized with pre-defined number of robots and a background bitmap image. The environment also adds one additional robot which serves to simulate obstruction in the parking space. This “obstruction” robot can be manipulated using a joystick peripheral.

5.6.1 Communication protocol

The communication between the Eiffel program and the simulation wrapper is conducted using simple ASCII strings. On connecting to the TCP socket (Port 54321), the wrapper sends continuous pose updates of the robots present in its environment (except the “obstruction” robot) as shown in Figure 14. The format of this update is as follows:

```
poses [number of robots] [robot id] [X coordinate] [Y coordinate] [Angle] [IR Sensor FL] [IR Sensor FFL] [IR Sensor FFR] [IR FR] ...
```

(The text in ***bold-italic*** is repeated for each robot)

Each robot has a unique id starting from 0. The robot has four Infra-red (IR) sensor to sense obstruction. The location of the IR sensors are shown in Figure 15. The IR sensors each return a value between 0 to 12, which indicates the distance to the obstruction in pixels.

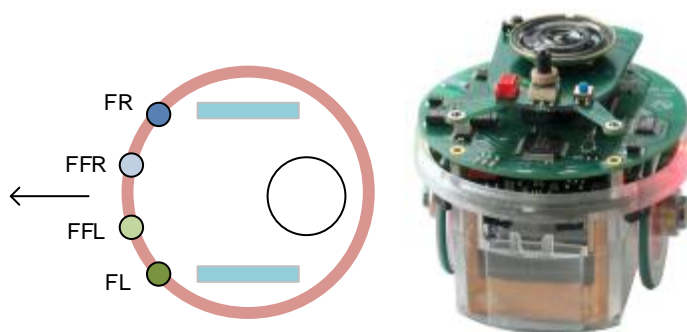


Figure 15: The location of the IR Sensors (the arrow shows the direction of motion when both wheel speeds are positive and equal). Photo from www.e-puck.org.

The angle of the robot, measured in radians, is construed as shown in Figure 16:

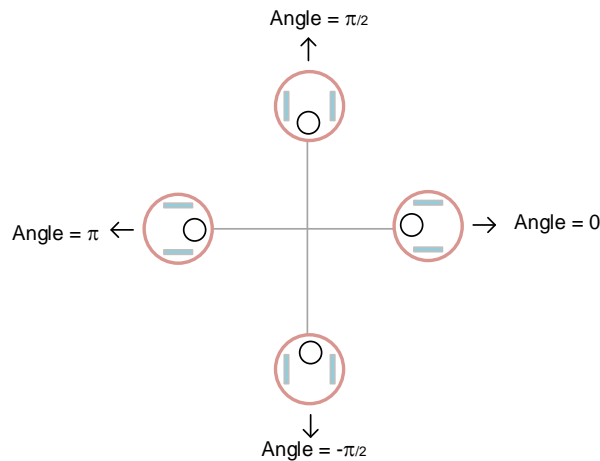


Figure 16: The interpretation of angle information

The robots can be commanded by sending a string constructed in the following format:

```
set [number of robots to command] [robot id] [left wheel speed] [right wheel speed]
[colour value red] [colour value green] [colour value blue] ...
```

The velocity of each wheel can be varied between -20 to +20. The robot has a ring of RGB LEDs and each of these three LEDs can be individually switched on or off by giving the *colour value* a 0 (off) or 1 (on).

5.7 Interface to Eiffel program

Communication to the simulator is established by using `NETWORK_STREAM_SOCKET` instance in the Eiffel program. Once the socket connection is made, we receive the pose updates (every 100 ms) in the network buffer. We need to read the buffer continuously so that we hold the latest pose information – to do this, we employ a helper routine which will continuously read the network buffer (Implemented in the class `LINK_POLLER`).

6. Comparison to traditional multi-threaded approach

In order to study the comparison between SCOOP and a commonly used concurrency programming methodology, a program using a multi-threaded approach was written based on the Microsoft .NET™ framework in C# language to achieve the mission control functionality. To draw parallels with the implementation in SCOOP (described in section 5), we now examine the implementation in C# by looking at the key parts – i.e. navigation, obstruction handling and low-level wheel control loop.

6.1 Navigation

For locking vertices on the path prior to navigation, we incorporate a mutex on each vertex (using the system class `AutoResetEvent`). First, the lock on the mutex is obtained by iterating through the vertices in the order of their ids, following which the navigation commences. Once a vertex has been traversed, the lock on its mutex is released.

```
public class Car
{
    private BackgroundWorker mWorkerThread = new BackgroundWorker();
    public Car()
    {
        mWorkerThread = new BackgroundWorker();
        mWorkerThread.DoWork += new DoWorkEventHandler(mWorkerThread_DoWork);
    }
}
```

```

public void move_to(int target_vertex_id)
{
    if (mWorkerThread.IsBusy)
        return;
    else
        mWorkerThread.RunWorkerAsync(target_vertex_id);
}

private void mWorkerThread_DoWork(object sender, DoWorkEventArgs e)
{
    int target = (int)e.Argument;

    List<int> pathIds = Map.get_path_to_target(target);
    List<int> sortedPathIds = Map.get_path_to_target(target);
    sortedPathIds.Sort();

    List<Vertex> verticesSorted = get_vertex_list(sortedPathIds);
    foreach (Vertex v in verticesSorted)
    {
        v.Mutex.WaitOne();
    }

    List<Vertex> verticesOnPath = get_vertex_list(pathIds);
    foreach (Vertex v in verticesOnPath)
    {
        move_to_vertex(v);
        v.Mutex.Set(); //Signals so that next waiting thread gains access.
    }
}

```

On the face of it, the above code snippet looks very similar to variant-1 of implementation in SCOOP, where we set flags in a sorted list of vertices and then started navigation. We now examine the conceptual differences in detail.

6.1.1 Safety

It is clear that other than the mutex on the vertex there is no other protection which prevents its use in case of concurrent access – i.e. **any client which foregoes the use of the mutex is able to access the vertex**. Also, an object instance which is executing an asynchronous action can still be accessed by a caller causing a possible modification in the object's state. For example, the car could be executing the navigation presuming the values of state variables related to its speed regulation and obstruction sensing, whereas during this period these variables are liable to be modified by a caller. To prevent access to the car during its execution of navigation, we would have to secure all methods on the car using monitors. Hence, compared to SCOOP, thread-safety has to be carefully planned and implemented in the code. On the other hand, use of synchronization primitives like the mutex gives the programmer a certain flexibility in coding. The mutex serves as a marker for resource reservation as well as providing wait semantics in one entity. The mutex and other synchronization entities in Microsoft .Net™ programming languages provide the programmer with conveniences like flexibly determining who can lock/unlock a mutex, how many threads can enter a mutex and how long to wait. For example, it is not difficult to implement a simple deadlock rescue mechanism by having a timeout on the mutex wait call (followed by a random wait and retry mechanism). Also, constructs like *WaitAll* (which causes the execution to wait until all in a collection of mutexes are signalled) allow simplified coding of synchronization. It is however important to note that with the use of synchronization entities as an augmentation to the actual resource we do not secure the resource itself. This factor is crucial to

consider when we wish to reason about safety, which is one of the key requirements in robotics programming.

6.1.2 Wait conditions

In multi-threaded programming, when a routine needs to access a shared resource, it would usually try and acquire a mutex associated with that resource (by waiting, if required). On the other hand, wait conditions and pre-conditions merge seamlessly in SCOOP. For example, consider the pre-condition to access a vertex is that it is not reserved or already occupied by another car. In case of SCOOP, this contractual condition automatically turns into a wait condition. Also, the wait behaviour of pre-condition in SCOOP is particularly useful when the predicate evaluation is based on multiple separate entities (in multi-threaded programming, this would have incurred multiple mutexes).

6.1.3 Asynchronous actions

In the code snippet listed in section 6.1, we return the call to `move_to` without any action if the worker thread is already busy. One way of allowing callers to queue up their requests is to use a mutex which allows only one caller thread to cause the navigation. This is shown in the code below:

```
public class Car
{
    public void move_to(int target_vertex_id)
    {
        mWorkerCompleted.WaitOne(); //Callers to move_to will have to wait here.
        mWorkerThread.RunWorkerAsync(target_vertex_id);
    }

    private void mWorkerThread_RunWorkerCompleted(object sender,
                                                    RunWorkerCompletedEventArgs e)
    {
        mWorkerCompleted.Set();
    }
}
```

In the above code, the caller would wait on the mutex `mWorkerCompleted`. Though this allows requests to queue, it requires this explicit semaphore guarding the routine. Again, here we see a clear benefit in SCOOP wherein the caller will wait till it acquires lock on the supplier's processor – hence if the car is busy with the navigation task, the parking manager will wait to obtain the lock on the car. The other benefit of using SCOOP is that the routine is executed asynchronously without having to deal with creation and starting of threads. Also, in SCOOP one could wait for the completion the asynchronous routine by making a query (or a command with result) which will result in join-like semantics.

6.2 Handling obstruction

In our program written in C#, we have the option of handling obstructions either as exceptions or events. We chose to use the event mechanism to demonstrate its elegance in understanding the control logic. The following code snippet shows that in case of a long-term obstruction the car will raise the event `Blocked`, and all subscribers to this event would be notified of this state. In our case the parking manager is the subscriber for this event.

```
public event EventHandler Blocked;

private void move_to_vertex(Vertex v)
{
    bool reached = false;

    while (!reached)
```

```

    {
        Pose currentPose = SimulatorLink.GetCurrentPose(this.Id);
        if (currentPose.IsObstructed)
        {
            State = CarState.Obstructed;

            if (ObstructionDuration.Seconds > 10)
            {
                if (Blocked != null)
                    Blocked(this, new EventArgs());
            }
        }
        else
        {
            Tuple<double, double> nextSpeed = GetNextWheelSpeed(currentPose);
            SimulatorLink.SetWheelSpeed(nextSpeed);
        }
    }
}

```

The parking manager then notifies all its cars to stop. It is able to do this because it can access the cars even if they are busy in their control loop execution.

```

public ParkingManager()
{
    foreach (Car car in mCars)
    {
        car.Blocked += Car_Blocked;
    }
}

private void Car_Blocked(object sender, EventArgs e)
{
    foreach (Car car in mCars)
    {
        car.stop();
    }
}

```

In Microsoft .Net™ event mechanism, objects interested in knowing about state change in the publisher subscribe themselves with the publisher by passing a reference to a call-back function. The publisher then notifies the subscribers by using the registered call-back functions. In Eiffel, an equivalence can be found in the mechanism of agents. However, the invocation of agent call on separate objects is subject to the same rule as routine calls [17]. Hence in regards to event based control of mission activities, we see certain clarity when using the event mechanism of Microsoft .Net™ (in particular, we see that it merges well with the object model). However, one has to be aware of the fact that the event call-back happens on the thread of the publisher. For example, the call to each `car.stop()` will be synchronously run on the thread of the car which raised the event. In case the call to `stop()` needs to be executed asynchronously, it will then be up to the programmer to implement the `stop()` action in the car in yet another background thread.

6.3 Control loop

The control loop to compute the next wheel speed is algorithmically same as the one in SCOOP version. **The performance of the control loop was observed to be much better than in the SCOOP version – this is attributed to overhead in SCOOP handler when attempting to lock and obtain current pose from the SIMULATOR_LINK entity.** For example with five cars in concurrent motion, it was possible for the

control loop cycle time in the C# program to be nearly 10ms whereas the shortest achievable cycle time in SCOOP was about 80ms.

7. Evaluating the application of SCOOP to mission control scenarios

Though the current implementation in this project does not contain a full-fledged mission control feature, it nevertheless serves to introduce some of the key coordination and concurrency issues which form the core requirements of mission control for autonomous vehicles in general. An example of implementation of mission control system described in [14] centers around separate entities handling activities such as system level organization and individual vehicle control. The framework proposed therein also deals with messaging and coordination between the entities. Mission control, in essence, is centralized in planning and coordination as opposed to the alternate model of decentralized control and coordination amongst autonomous robots. In this project, the focus is on centralized mission control architecture. Here, the paradigm of SCOOP with its notion of separateness maps well toward providing a solution.

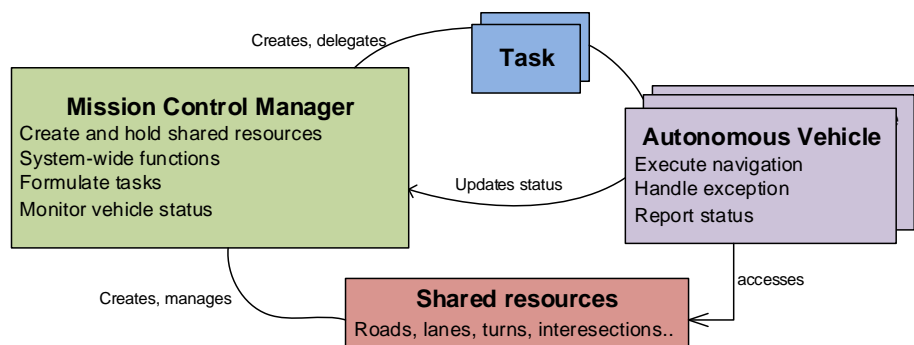


Figure 17: An abstract view of components involved in mission control systems

One of the main tasks of a mission control is the creation, delegation and monitoring of tasks. An orchestrating entity, such as the parking manager in our implementation, is responsible for centralized creation of tasks (see Figure 17). A task might be composed of sub-tasks and can have dependencies to other tasks. The tasks in turn are executed by autonomous elements like the cars, which are not necessarily in the same computational context as the orchestrating entity (the Mission Control Manager). For example, the autonomous entities could have their own CPU on which the delegated task is processed and executed. Such entities can then manage the execution of the task using mechanisms like event-based control or finite state machinery based on petri-net model. One proposed implementation for large scale autonomous vehicles called AORTA [16] suggests use of execution agents which encapsulates behaviour and strategy allowing the vehicle to make decisions locally and with minimal interaction with the mission control.

Taking an abstract view of our implementation, we see the following requirements of a hypothetical mission control framework towards the programming and runtime paradigms:

- Messaging between entities which are not necessarily in the same process space.
- Safe access to shared resources with guaranteed mutual exclusion and race-free condition.
- Indication, if not prevention, of dead-lock condition.
- Exception handling and recovery mechanism

7.1 Messaging between entities

Though presently limited to a single process boundary, messaging between objects on distinct logical processors is well handled in SCOOP. The separate objects may interact either by calling routines on the each other or use Eiffel Agents [17] effectively. There is also an ongoing research effort to extend this to a truly distributed environment wherein separate objects are handled by different processors across machine boundaries.

7.2 Safety

Coupled with the ability to execute routines on separate objects, the safety aspect is carried along seamlessly in SCOOP. Routine calls on separate objects can only be executed when the caller owns the lock on the handler of the target object. Apart from guaranteeing exclusive access, this ensures that race conditions do not occur when a shared resource is concurrently accessed by two concurrent consumers.

7.3 Deadlocks

When dealing with separate entities, we are still liable to run against dead-lock situations in SCOOP if the sequence of obtaining locks is not take care of. To address this critical problem which is prevalent in concurrent programming in general, there is a solution proposed in [15]. This is yet to be implemented in SCOOP.

7.4 Exception handling

Perhaps the most critical deficit in SCOOP that was realized during this project was the absence of any form of exception handling in asynchronous calls. In the current implementation, such exceptions would fail silently. This also poses a limitation when one wishes to design control based on exceptions – for example it would quite natural to handle obstruction of a vehicle as an exception and let the caller (the parking manager, in this case) take alternative rescue action. The mechanism proposed in [10] suggests that an exception occurring during an asynchronous routine execution in the supplier is propagated to the caller only when it holds the supplier accountable. If we hypothetically consider the availability of this mechanism, we see that one such point where the mission control, which is the caller, can hold the vehicle accountable is on query of car status. However, mission control architectures which use a supervisory object to monitor the vehicles would not benefit from this mechanism as the accountability expires with the lock. The following pseudo-code explains the usage scenarios:

Using query to receive exception	Using a supervisory object
<pre>class MISSION_CONTROLLER supervisor : separate CAR_SUPERVISOR send_tasks (c1..cn: separate CAR) c1.move_to(v1) ... cn.move_to(vn) s := c1.status -- can throw! supervisor.supervise()</pre>	<pre>class CAR_SUPERVISOR cars : LINKED_LIST[separate CAR] supervise foreach c in cars do get_car_status(c) get_car_status(c : separate CAR) c.status -- will not throw!</pre>

7.5 Summary

We see that use of SCOOP in mission control of autonomous vehicles brings significant advantages in terms of safe access to mutually exclusive resources, asynchronous execution and wait conditions. The lack of exception handling however makes error handling in the control sequence to be cumbersome.

And also, support for true distributed hosting of separate objects would prove to be invaluable when considering multiple robots, each with its on-board processing capability.

8. Conclusion

In the realm of robotic control programming, one often has to deal with shared resources which are also tangible physical components in reality. In autonomous systems, such components are capable of executing tasks asynchronously and independently. The object-oriented representation of such components does not model or enforce the aspect of the component being a shared resource and its possible constraint of exclusive usage. Traditionally, mutual exclusion and asynchronous execution requirements are overlaid on the object model by using concepts like multi-threading and synchronization primitives. We have seen in this implementation of the automated parking that the paradigm of separate objects in SCOOP maps well with the way tangible entities are handled in the real world.

For autonomous execution of tasks, as in the navigation of a car, there are both advantages and challenges in SCOOP compared to the traditional approach of multi-threading. We have seen that there are no additional constructs required in SCOOP to effect the asynchronous navigation action on the car. The access to a separate object is protected by the requirement that the caller needs to own the lock on the processor, thereby ensuring safety in the event of concurrent attempt to access the object. However, the rather stringent safety mechanism also sometimes poses inconvenience in design. One such case is when an asynchronous autonomous action has been invoked (like the *move_to(target)*), the processor of the car is busy executing it in a closed loop control. There is then no possibility for a caller, including the one which invoked the asynchronous action, to modify the state of the car. This, for example, is required when the parking manager needs to instruct all cars to stop. Here, we had implemented code in the car which polls for status indicating stop request from the parking manager, whereas having the possibility to set a state variable seemed to be more natural. Similarly, handling of exceptions in asynchronous executions is yet another topic that presents a gap – currently the exception causes a silent failure and the caller is not notified. It would be interesting to examine these issues from the perspective of robotic programming while developing the future roadmap of SCOOP.

In conjunction with the implicit asynchronous nature of commands, there is also the complementary synchronous nature of queries (which also waits for the previous asynchronous commands to complete). These two mechanisms enable caller to invoke an asynchronous operation on a component and then later wait for its completion. Finally, the role of pre-condition evaluation on separate objects turning into wait condition fits in naturally to scenarios where the service provided by a component of the robot or an autonomous entity is dependent on the state of another component. The use of SCOOP in solving few other robotic control problems are described in [4] [7] and [11] – in all these studies, the advantage of these features has been highlighted.

The natural mapping between real world entities in robotics and the notion of separate objects brought about in SCOOP displays its benefits when we explore concrete problems in robotics. Guaranteed race-free conditions, asynchronous execution of commands, the need to obtain lock on the target object's processor before use, and pre-conditions functioning wait conditions are indeed interesting and useful when implementing robotic controls. With the increasing interest of the robotics programming community in exploring alternative paradigms for concurrency, SCOOP offers several benefits in addition to the “design-by-contract” philosophy in Eiffel.

9. References

- [1] The V-Charge project : <http://www.v-charge.eu/>, [Online]
- [2] B.Meyer, "Systematic Concurrent Object-Oriented Programming," Communications of the ACM, vol. 36, no. September, pp. 56-80, 1993.
- [3] P.Nienaltowski, "Practical framework for contract-based concurrent object-oriented programming," PhD dissertation 17061, Department of Computer Science, ETH Zurich, February 2007.
- [4] G.Ramanathan, B.Morandi, S.West, S.Nanz and B.Meyer, "Deriving concurrent control software from behavioural specifications," IEEE/RSJ International Conference on Intelligent Robots and Systems, pages 1994-1999.
- [5] Coffmann, Shoshani, "System Deadlocks," ACM Computing Surveys, pp. 67-78, 1971.
- [6] Eiffel, "<http://docs.eiffel.com/book/solutions/concurrent-eiffel-scoop/>," [Online].
- [7] Enki, "<http://home.gna.org/enki/>," [Online].
- [8] M.Ben-Ari, "Principles of Concurrent and Distributed Programming", Prentice Hall Europe, 1990.
- [9] G.Andrews, "Concurrent Programming - Principles and Practice", Addison-Wesley, 1991.
- [10] B.Morandi, S.Nanz and B.Meyer, "Who is accountable for asynchronous exceptions?" in Asia-Pacific Software Engineering Conference, 2012.
- [11] G.Ramanathan, "SCOOP for Robotics - Implementing bio-inspired hexapod locomotion," Project Report, ETH, Zurich, 2009.
- [12] B.Meyer, "Object Oriented Software Construction", Prentice Hall, 1997.
- [13] B.Morandi, S.Nanz and B.Meyer, "A Formal Reference for SCOOP", Empirical Software Engineering and Verification, volume 7007 of Lecture Notes in Computer Science, pages 89-157. Springer, 2012.
- [14] P.Oliveira et. al, "On the design and development of mission control systems for autonomous underwater vehicles", Proceedings of the 6th International Symposium on Robotics and Automation (ISRAM'96), 1996
- [15] S.West, S.Nanz and B.Meyer, "A modular scheme for deadlock prevention in object-oriented programming model", Proceedings of the 12th International Conference on Formal Engineering Methods (ICFEM'10), Springer, 2010.
- [16] Carlino et al, "Approximately orchestrated routing and transportation analyser: large-scale traffic simulation for autonomous vehicles", Proceedings of 15th IEEE Intelligent Transportation Systems Conference, September 2012.
- [17] Agents in SCOOP, "http://dev.eiffel.com/Agents_in_SCOOP/", [Online]
- [18] S.Nanz, F.Torshizi, M.Pedroni and B.Meyer, "Design of an Empirical Study for Comparing the Usability of Concurrent Programming Languages", Information and Software Technology, 55(7):1304-1315, Elsevier, 2013.
- [19] Microsoft MSDN Help "[http://msdn.microsoft.com/en-us/library/System.Threading.WaitHandle\(v=vs.110\).aspx](http://msdn.microsoft.com/en-us/library/System.Threading.WaitHandle(v=vs.110).aspx)" [Online].

10. Appendix-A: Setting up the simulation environment

The steps required to get the simulation environment running on a Linux machine is described here. The version of Linux on which these steps were carried out was “Mint 3.8”, but however these should be equally valid for other flavours of Linux too.

A. First, we need to get some essential libraries. Open the terminal window and execute the following commands:

```
sudo apt-get install cmake
sudo apt-get install libqt4-dev
sudo apt-get install libgtk1.2 (now libgtk2.0)
sudo apt-get install libiw-dev
sudo apt-get install libhal-dev
sudo apt-get install build-essential
sudo apt-get install git
sudo apt-get install libudev-dev
sudo apt-get install libssl1.2-dev
```

B. We can now fetch and compile Enki which is the robot simulation framework (updated version: <https://github.com/enki-community/enki>)

```
sudo git clone https://github.com/enki-community/enki
sudo cmake .
sudo make
sudo make install
```

C. Fetch and compile dashel which is a data stream helper library

```
git clone git://github.com/aseba-community/dashel.git
sudo cmake .
sudo make
sudo make install
```

D. Finally, fetch and compile the simulator wrapper

```
sudo git clone https://github.com/ethz-asl/displayswarm-sim
sudo cmake .
sudo make
```

E. We are now ready to run the simulator. The command usage is:

```
dss <number of robots> <optional:background bitmap>
```

For example the following command will launch the simulation viewer with three e-puck robots and the PATH.png image as background

```
./dss 3 PATH.png
```

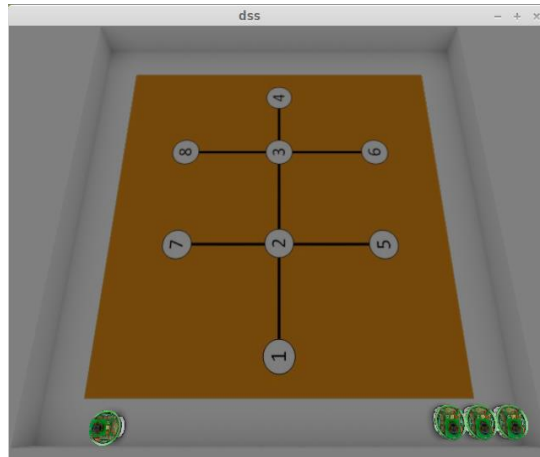


Figure 18: The viewer of the simulator

The three robots are placed on the wall side. One additional robot is created which serves to simulate obstruction. If a joystick is connected to the machine, then this robot can be manipulated with the joystick.

Viewport manipulation

The viewport of the simulation window can be manipulated with combination of key press and mouse drags:

Ctrl + Left mouse button pressed + drag: Changes the viewing angle.

Ctrl + Shift + Left mouse button pressed + drag: Offsets the view in the windows extents.

Ctrl + Shift + Right mouse button pressed + drag: Changes zoom level.

(Note the view setting are not persisted).

Connecting a client to the simulator

To see the TCP packets use any telnet client (or TCP client with ASCII string terminal dump) and connect to port 54321 (IP address of the machine on which the simulator is running. Please make sure you firewall policies don't block this port).

```
telnet localhost 54321
```

Once connected you will receive pose updates in the format:

```
poses [number of robots] [robot id] [X coordinate] [Y coordinate] [Angle] [IR Sensor FL] [IR Sensor FFL]  
[IR Sensor FFR] [IR FR] ... (The text in bold is repeated for each robot)
```

You can also command the robots with following syntax:

```
set [number of robots to command] [robot id] [left wheel speed] [right wheel speed] [color value red]  
[color value green] [color value blue] ...
```

The wheel speeds can be varied from -100 to 100. The robot has a ring of RGB LEDs and each of these three LEDs can be individually switched on or off by giving the color value a 0 (off) or 1 (on).

11. Appendix-B: Setting up the Eiffel project “scoop_park”

Obtain the project sources and restore it on your local drive – all required setting to enable SCOOP are already in place. The code needs two configuration values to be provided – the IP Address of the machine where the simulator is running and the location of the graph file. The IP Address is specified in the SIMULATOR_LINK

```
POSE SIMULATOR_LINK PARKING_MANAGER VERTEX MAP CAR
feature
  robot_count : INTEGER

  open
    do
      create socket.make_client_by_port(54321, "192.168.0.112")
      socket.connect
    end
```

Next, in the code for the class MAP, specify the location of the graph file:

```
POSE SIMULATOR_LINK PARKING_MANAGER VERTEX MAP CAR
read_graph_file
  local
    str:STRING
    count, i, j, k:INTEGER
    parts: LIST[STRING]

    neighbors : LINKED_LIST[INTEGER]

    x,y:DOUBLE
    v: VERTEX
  do
    create input_file.make_open_read ("c:\temp\graph.txt")
    create vertices.make
    create edges.make

    input_file.read_line
    str := input_file.last_string;
    str.prune_all ('%R')
    count := str.to_integer_32
```

The graph file is a simple text content describing the vertices and the adjacency list. The format of the file content is:

The first line contains the count of vertices

[Number of vertices]

Then for each vertex, specify the X and Y coordinates (in pixels) separated by semi-colon character:

[X];[Y]

Then for each vertex list the adjacent connected vertices:

[Vertex id, starting with zero];[neighbors, separated with ;]

Here is an example graph file for the following layout (**the size of the bitmap should be 100 x 100 pixels, or it will be re-scaled by the viewer**). The sources for this example layout are stored in the project deliveries.

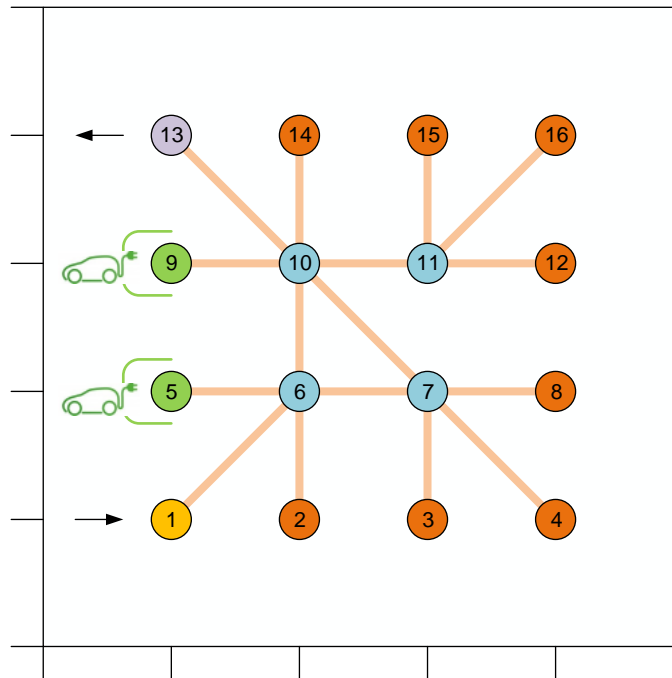


Figure 19: An example path graph with grid size of 20 pixels

16	1;6;
20;20	2;6;
40;20	3;7;
60;20	4;7;
80;20	5;6;
20;40	6;7;5;10;2;1;
40;40	7;3;10;6;8;4;
60;40	8;7;
80;40	9;10;
20;60	10;9;13;14;11;6;7;
40;60	11;10;15;16;12
60;60	12;11;
80;60	13;10;
20;80	14;10;
40;80	15;11;
60;80	16;11;
80;80	

The layout boundaries needs to be a square – the actual dimensions do not matter as the bitmap is anyway resized to 100 x 100 pixels. In order to retain a good quality after rescaling, it is recommended that the file should be saved as PNG (Portable Graphics Notation).

Once the above configurations are put in to the code, you can start the application console:

```

Welcome to SCOOP PARK!
-----
Please provide test instruction sets.
Each instruction is a set of car id and target vertex id.
To end the instructions: 0 <Enter>
-----
Car Id <0 to end>:

```

The default console allows you to test navigation commands. You can enter set of instructions which will then be fired off to the cars for concurrent execution. Once the execution starts, you can see the debug messages:


```

Welcome to SCOOP PARK!
-----
Please provide test instruction sets.
Each instruction is a set of car id and target vertex id.
To end the instructions: 0 <Enter>
-----
Car Id <0 to end>:1
To vertex:6
Car 1 will move to vertex 6
Car Id <0 to end>:2
To vertex:7
Car 2 will move to vertex 7
Car Id <0 to end>:

```

```

-----
Car Id <0 to end>:1
To vertex:6
Car 1 will move to vertex 6
Car Id <0 to end>:2
To vertex:7
Car 2 will move to vertex 7
Car Id <0 to end>:0
Executing instructions
Car id <0 to end>:Car:0: estimated node is: 5
Car:1: estimated node is: 8
Computed path is:5 -> Computed path is:20 -> -> 3 -> 3 -> 6 -> 2 ->
7 ->
move_to_vertex called50,30
Car id 0 instructed to move to 50,30
Car id 0 completed last instruction
move_to_vertex called50,60
Car id 0 instructed to move to 50,60
Car id 0 completed last instruction
move_to_vertex called80,60
Car id 0 instructed to move to 80,60

```

The navigation algorithm variant to use can be set in the code for the class CAR:

```

POSE SIMULATOR_LINK PARKING_MANAGER VERTEX MAP CAR APPLICATION
end

feature --Behaviour

--The distance limit between the car and an obstruction below which the car will b
obstruction_tolerance : DOUBLE assign assign_obstruction_tolerance

--Routine to cause the car to navigate autonomously to the target node.
move_to_target(target_vertex_id : INTEGER)
  local
    current_vertex_id, start_vertex_id : INTEGER
    j : INTEGER
    variant_to_use : INTEGER
  do
    variant_to_use := 2 -- Set which variant (described in report) to use.

    clear state

```

Simulating obstruction in absence of joystick hardware

In case joystick is not available to move the “disturber” robot on the simulation viewer, one of the normal cars can be designated as the disturber by setting its variable `is_disturber` to true. Once this is done, this car can be moved to any vertex without having to wait for locks.