# Simulation d'une voiture autonome sur le RTOS QNX

Antonin Godard (2032402)
*Génie Informatique*
*Polytechnique Montréal*
Montréal, Québec
antonin.godard@polymtl.ca

Rayan Neggazi (2038882)
*Génie Informatique*
*Polytechnique Montréal*
Montréal, Québec
rayan.neggazi@polymtl.ca

***Résumé*—Cet article présente la simulation d'une voiture autonome sur un système d'exploitation temps-réel QNX (Blackberry). Cette voiture se déplace sur un terrain supposé plat, où elle doit atteindre une destination finale en passant par plusieurs étapes. Elle doit également prendre en compte des obstacles générés aléatoirement sur sa trajectoire.**

**L'étude se focalise sur la modélisation de la partie continue ainsi que la partie discrète du système. Il ne s'agit donc en aucun cas d'étudier les mécaniques de la voiture ou l'électronique des composants. L'article porte sur la modélisation de la partie « intelligente » du système et principalement trois entités :**

**— le contrôle de la destination de la voiture ;**
**— le contrôle de la navigation de la voiture ;**
**— le contrôle de la caméra de la voiture.**

**Cet article montre les différentes techniques de parallélisation de tâches ainsi qu'un exemple de partitionnement CPU de ces dernières. On parlera aussi des cas d'utilisation ajoutés au système en vue de son amélioration.**

**Il y figure également les problèmes rencontrés ainsi que les solutions apportées et les résultats associés à ces solutions.**

***Index Terms*—simulation, RTOS, QNX, partionnement, modélisation**

## I. Introduction

La voiture autonome a été premièrement analysée et décrite avec les différentes normes UML. Nous avons choisi de réaliser quatre diagrammes (voir Annexe) :

— un diagramme de cas d'utilisation afin de décrire dans quel contexte évolue le système ;
— un diagramme de séquence qui décrit le comportement du système dans le temps ;
— un organigramme décrivant la partie logique du système
— un diagramme d'objets pour décrire la structure de la voiture.

Ces quatre diagrammes nous ont permis d'établir une vision claire du système afin de passer à la modélisation et à la simulation.

Dans un premier temps, nous avons réalisé une modélisation avec TrueTime, un simulateur s'exécutant sur MatLab et Simulink. Dans cette partie, nous avons réalisé la partie *discrète* du système – c'est à dire la modélisation du comportement de la voiture. Cette partie discrète envoie des commandes à la partie continue et de cette façon nous contrôlions la voiture (sa vitesse, son orientation, et la prise de photo).

Dans un second temps, nous sommes passé à la simulation de la voiture sur une autre plateforme : le RTOS QNX, développé par Blackberry. Ce système d'exploitation temps-réel permet l'implantation de systèmes temps-réels avec l'utilisation de mécanismes sophistiqués régis par la norme POSIX.

Cette simulation s'est effectuée en deux temps :

1) une modélisation purement fonctionnelle de la voiture ayant pour objectif de répondre au cahier des charges préalablement fourni
2) une amélioration du modèle ainsi que l'ajout de mécanismes propres à QNX

Ainsi, l'article portera sur la dernière partie de la modélisation sur QNX. On parlera des problèmes rencontrés, des solutions proposées et des résultats obtenus afin d'obtenir un système performant.

Pour modéliser la voiture, nous avons séparé les différentes fonctions de la voiture en tâches qui communiquent à travers des variables. Chaque tâche est en fait un fil d'exécution de la bibliothèque `pthread`. Tout ces fils sont initialisés au même instant et ont chacun un second fil associé représentant leur timer, et donc leur périodicité.

Plus précisément, chaque tâche de la voiture a une routine infinie associée, qui est lancée depuis une routine principale nommée `main_worker` qui leur est toute commune. Ce choix d'avoir une routine commune à chaque fil qui vient lancer une routine spécifique en fonction du fil rend le code plus lisible mais surtout plus modulaire. Il nous était donc facile de modifier les caractéristiques d'une tâche (période, priorité), d'en ajouter ou d'en supprimer.

La séparation des tâches respecte la description initiale de notre système. Les tâches continues et discrètes ont chacune un fil d'exécution associé. Nous avons ajouté une tâche d'affichage (display), deux tâches de détection de niveau de batterie (faible et élevé), et une tâche de trace de données, sur laquelle nous reviendrons dans la section III.

## II. Lacunes de la première implantation

### A. Incertitude de la périodicité des tâches

Chaque tâche dans notre programme a besoin d'être exécutée de manière périodique, avec des périodes différentes

évidemment. Afin de remplir cette fonction, nous avons initialement intégré des instructions de type « sleep » dans chaque tâche avec le temps pendant lequel nous souhaitions endormir notre programme.

Toutefois, les tâches n'étaient pas réveillées de de manière précise a chaque intervalle de temps indiqué par l'instruction sleep en question. En effet, le programme s'endort bien pour cette période mais seulement après avoir exécuté toutes les autres instructions qui le composent. Ainsi, la période réelle de la tâche est le temps de consigne du sleep auquel il faut ajouter le temps d'exécution de l'ensemble du programme qui de plus, varie d'un appel à l'autre en fonction des conditions et des paramètres actualisés.

### B. Variables globales pour la synchronisation

Étant donné la nécessité de communiquer entre les parties *opérative* et *contrôle* notamment, nous avons ,dans un premier temps, fait le choix d'utiliser des variables globales.

Cependant, cette méthode ne garantissait ni un accès optimal aux variables partagées, ni une synchronisation parfaite entre les threads de contrôle et d'opération.

### C. Temps de simulation

Après avoir terminé la modélisation de base de la voiture, la simulation fonctionnait mais elle ne pouvait être accélérée.

La simulation se réalisait en temps réel comme si la voiture avançait réellement à 30 ou 50km/h. Il fallait attendre trop longtemps pour que la voiture atteigne un nombre signifiant d'étapes, de stations et de destinations. De plus, la batterie se déchargeait trop lentement.

## III. Solutions et améliorations proposées

### A. Solutions aux lacunes

*1) Ajout de timers:* Afin de corriger cette erreur causée par l'utilisation de la fonction sleep dans nos tâches, nous avons remplacé ces derniers par des timers. Les fonctions nécessaires à cet objectif ont été extraites d'un code fourni, et intégrées à part dans un fichier timers.cpp.

Ainsi, pour chaque tâche nous avons initialisé un timer (grâce à la fonction init_timer) prenant en paramètre notamment la période désirée, ainsi qu'un pulse handler (à l'aide de la fonction task_pulse_handler). Celui-ci permet alors de réveiller la tâche en question à chaque période par l'utilisation d'une sémaphore. Le thread, pour être exécuté doit désormais attendre que sa sémaphore soit relâchée par son pulse handler.

*2) Synchronisation des tâches aves des sémaphores:* Pour exécuter les fils périodiquement nous utilisons un fil associé à chaque thread qui modélise un timer. Ce fil vient périodiquement relâcher une sémaphore sur laquelle chaque tâche attend. Ces sémaphores nommées `sync_sem` font partie de la structure de données passées à chacune des tâches.

Pour la synchronisation de la partie contrôle de la caméra, nous utilisons également une sémaphore pour « réveiller » la partie continue de la caméra tous les 10 mètres. Deux sémaphores sont aussi utilisées pour réveiller les deux fils associés à la détection de batterie haute et faible.

*3) Accélération de la simulation:* Pour palier au problème de temps de simulation, nous avons ajouté un macro `SIMU_ACCEL` nous permettant d'ajuster la vitesse de simulation. Ce facteur vient jouer sur plusieurs paramètres :

- augmenter le pas d'avancement de la voiture ;
- accélérer la décharge la batterie ;
- multiplier le temps de trace de données pour avoir une estimation du temps réel écoulé.

Cependant, accélérer la simulation veut aussi dire qu'il faut ajuster la période des tâches ayant besoin de précision. Nous avons notamment rencontré un problème lors de la détection à moins de dix mètres.

Ainsi, les tâches de contrôle ont une fréquence bien supérieure aux tâches continues.

### B. Améliorations

*1) Utilisation raisonnable de mutex:* Il nous a semblé important d'utiliser le strict minimum en terme de mutex car ce sont des mécanismes utilisant beaucoup de ressources. Ainsi, nous en utilisons 4 pour les 4 données physiques de la voiture : vitesse, angle, niveau de batterie, et position courante. Nous pensons que cela est un bon compromis entre le nombre de mutex utilisés et la granularité des accès.

En effet, nous aurions pu n'utiliser qu'une seule mutex pour les quatre données (toutes stockées dans la même structure) mais cela aurait bloqué trop de tâches.

*2) Trace de données:* Garder un historique des données de la voiture nous semblait être une bonne amélioration étant donné qu'en réalité les logs sont indispensables au débogage. Nous avons alors ajouté un fil d'exécution qui écrit périodiquement le données de la voiture dans un fichier toutes les 100ms.

Cette amélioration est également utile afin de montrer les résultats obtenus, dont nous discutons section IV. On peut notamment voir l'évolution de la vitesse et de la batterie en fonction de la position et du temps.

Le code servant à tracer les données se trouve en annexe.

*3) Ajout d'obstacles devant la voiture:* Pour le contrôle de la navigation, nous avions décidé de modéliser le comportement de la voiture par une machine à états, car cela augmente la modularité de l'implantation, et permet une meilleure compréhension du comportement.

Ainsi le résultat que nous avons obtenu est celui de la figure 1. L'état `GOTO_DEST` représente l'état initial de la voiture, où elle doit se diriger vers la prochaine étape. L'état `PRE_BATTLOW` survient lorsque la batterie est faible, met la vitesse de la voiture à 30km/h et détermine la station la plus proche. L'état `BATTLOW` fait en sorte de déterminer si la voiture est arrivée à la station. Lors de l'arrivée la voiture passe à l'état `CHARGING`, qui représente la recharge de la voiture. Finalement la voiture retourne à l'état initial lorsque la batterie est chargée.

Mais nous nous placions également dans une situation réelle où il est possible que la voiture rencontre des obstacles – que ce soit lorsqu'elle se rend à une étape ou à une station. Ainsi,

nous avons ajouté deux états à la machine afin de modéliser cet ajout.

La génération d'obstacle est aléatoire et s'exécute à chaque fois que la voiture atteint une étape. L'obstacle sera, avec une probabilité prédéfinie, positionné à une distance elle aussi aléatoire entre la voiture et la prochaine étape.

L'état `PRE_OBSTACLE` est alors atteint quand la voiture arrive à 5 mètres de l'obstacle. Cet état met la vitesse de la voiture à 0 et démarre un timer. En fait, en réalité, cette fonction de la voiture serait un capteur qui détecte et met en arrêt d'urgence la voiture lorsque une personne ou un objet est devant.

L'état `OBSTACLE` attend que le timer soit écoulé, puis fait repartir la voiture. Nous aurions pu modéliser cela de plusieurs façons mais nous avons jugé le timer suffisant. Également, cet état doit prendre compte la variable `lowBat`, représentant l'état de la batterie faible, pour retourner au bon état.

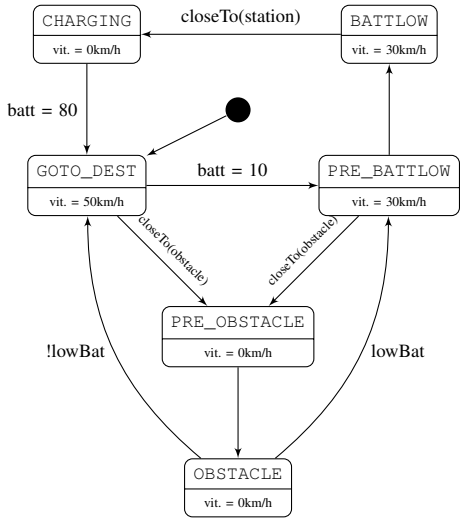



FIGURE 1. Contrôle de la navigation avec obstacle

*4) Partitionnement CPU des tâches:* Un des mécanismes de QNX que nous avons choisi est le partitionnement des tâches. QNX propose un partitionnement *adaptatif* qui permet l'assignation de tâches dynamiquement, après leur création. Le partitionnement est un point critique des systèmes temps-réel ayant besoin d'un haut niveau de sûreté. Cela permet notamment de diviser les tâches en groupes et de définir un *pourcentage* de temps CPU alloué à la partition.

Nous avons choisi de diviser les tâches continues et les tâches discrètes dans des partitions séparées car ce sont deux corps de tâches avec un fonctionnement différent. Cela se justifie également par l'aspect sécuritaire que le partitionnement ajoute : il limite l'usage des ressources CPU, ce qui empêche une partition gourmande de prendre toutes les ressources et de bloquer les autres tâches dans les autres partitions.

Nous avons choisi de limiter la partie continue à 20% ainsi que la partie discrète à 60%.

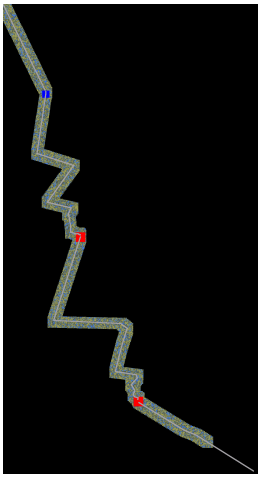


FIGURE 2. Tracé avec `takePhoto`

## IV. RÉSULTATS

Pour cette section, nous nous baserons sur une seule et même exécution de la simulation.

### A. Accélération de la simulation

L'accélération de la simulation a bien fonctionné. Comme peut en témoigner l'analyse de photo figure 2, on atteint bien deux étapes et une station.

### B. Trace des données de la voiture et analyse graphique

On peut observer le déplacement de la voiture figure 3. Nous nous sommes limités à l'affichage des deux premières étapes pour améliorer la lisibilité. La légende indique les points importants du graphique, et la barre de couleur indique l'état de la batterie à un point donné.

On peut donc confirmer que la voiture est bien chargée après le départ d'une station, et que la voiture effectue bien son chemin étape par étape jusqu'à la destination finale.

On remarque aussi qu'une étape proche de la station n'est pas atteinte. Cela s'explique pqr le fait que la voiture régénère une étape après être chargée.

La figure 4 montre l'évolution de la vitesse pendant la simulation. On voit que la voiture est à 30km/h pendant un bref instant pour aller à la station. On voit également que la voiture s'arrête pendant un instant lorsqu'elle détecte un obstacle.

### C. Partitionnement CPU

Les résultats obtenus s'observent à la figure 5, qui représente une simulation de 60 secondes.

On observe que la partition discrète consomme beaucoup plus que la partition continue. C'est normal, puisque la période de la partie discrète est 50 fois moins grande que celle des tâches de l'autre partition – nécessaire afin d'accélérer la simulation.

On remarque également que la partition système consomme plus de ressources au début afin de créer les fils, créer les partitions et assigner les précédents fils aux partitions. Sa consommation pendant la simulation s'explique car les timers

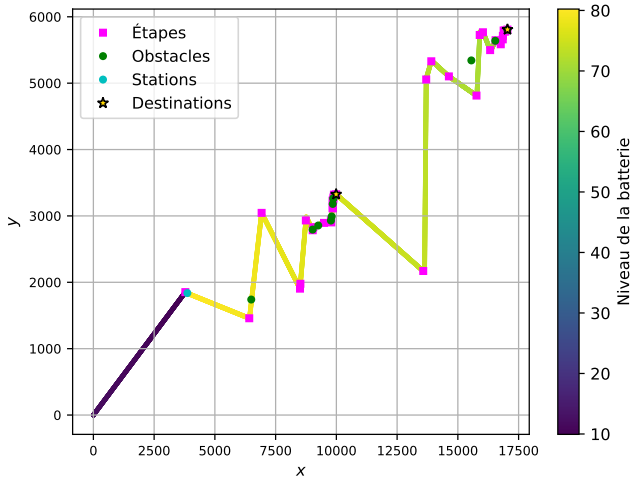



FIGURE 3. Chemin de la voiture

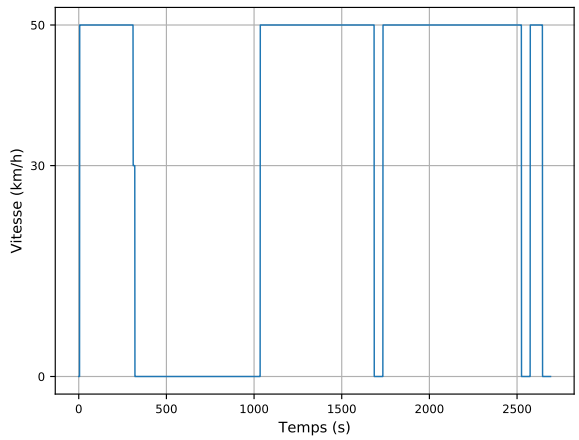



FIGURE 4. Évolution de la vitesse

tournent toujours sur la partition système et d'autres processus tournent également sur l'OS en parallèle de notre programme.

## V. CONCLUSION

L'objectif de ce projet était d'implémenter le comportement de la voiture autonome modélisée depuis son commencement, en prenant en compte les enjeux du concept temps-réel. Ainsi, le système d'exploitation QNX a été employé à bon escient, avec l'utilisation de tous les outils tels que les sémaphores, mutexs et timers qui nous ont permis de répondre aux exigences temporelles du système.

D'autre part, nous avions également pour objectif d'améliorer le système en termes de fonctionnalités et de performance. Parmi ces améliorations nous avons compté l'optimisation de l'utilisation des ressources CPU et mémoire du système ainsi que la modélisation correcte du comportement du système afin qu'il puisse faire face aux différentes situations possibles.

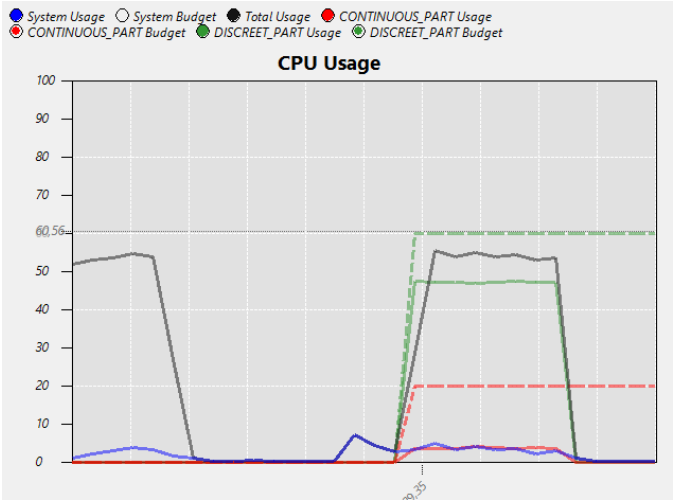



FIGURE 5. Partitionnement des tâches discrètes et continues

Nous pouvons dire que nous sommes parvenus dans un premier temps à apporter des solutions efficaces aux différents problèmes auxquels nous avons fait face dans la première étape du projet. Mais aussi, nous avons pu intégré des améliorations à notre travail qui permettent de représenter le fonctionnement de la voiture autonome de manière plus réaliste ainsi que d'optimiser l'implémentation du système.

La simulation, présentée dans cette article, témoigne de la performance du système, étant donné qu'elle a pu être exécutée de nombreuses fois avec facteur d'accélération assez important afin de garantir son bon fonctionnement sur un grand nombre de destinations atteintes.

Enfin, nous notons que d'autres améliorations auraient pu être ajoutés à notre travail, telles que l'affinité des threads (qui consiste à lier chaque thread à un CPU spécifique, utilisé généralement pour palier aux erreurs de cache) ou encore l'intégration de « watchdogs » (procédé souvent utilisé dans le temps réel, il permet d'éviter le blocage du processus en cas d'erreur ou d'action malveillante en redémarrant le système notamment).

# ANNEXE

Listing 1. ./code/plot_data.py

```
import matplotlib.pyplot as plt
import numpy as np

data_f = open("./data.csv", "r")
first = data_f.readline().replace("\n", "").split(",")
data = []
[data.append([float(el)]) for el in first]

length = len(first)

for line in data_f:
    line_list = line.replace("\n", "").split(",")
    [data[i].append(float(line_list[i])) for i in range(length)]

data_f.close()

def unique(seq1, seq2):
   checked1 = []
   checked2 = []
   last_el = None
   for i in range(len(seq1)):
       if seq1[i] != last_el:
           checked1.append(seq1[i])
           checked2.append(seq2[i])
           last_el = seq1[i]
   return checked1, checked2

print(data[12][3000])
print(data[13][3000])


for i in range(6,14,2):
    data[i], data[i+1] = unique(data[i], data[i+1])

print(data[12])

# Map plotting
plt.figure(1)
plt.scatter(data[4], data[5],c=data[1], s=4)
cbar = plt.colorbar()
cbar.set_label('Niveau de la batterie')
plt.plot(data[6], data[7], 's', c='fuchsia', markersize=4, label="Étapes")
plt.plot(data[10][1:], data[11][1:], 'go', markersize=4, label="Obstacles")
plt.plot(data[12][1:], data[13][1:], 'co', markersize=4, label="Stations")
plt.plot(data[8], data[9], '*', c="gold", mec="black", markersize=7, label="Destinations")
plt.xlabel("$x$")
plt.ylabel("$y$")
plt.xticks(fontsize=8)
plt.yticks(fontsize=8)
plt.legend()
plt.grid()
plt.show()

plt.figure(2)
plt.plot(data[0], data[2], lw=1)
plt.grid()
plt.xlabel("Temps (s)")
plt.ylabel("Vitesse (km/h)")
plt.xticks(fontsize=8)
plt.yticks(fontsize=8)
plt.yticks([0,30,50])
plt.show()
```

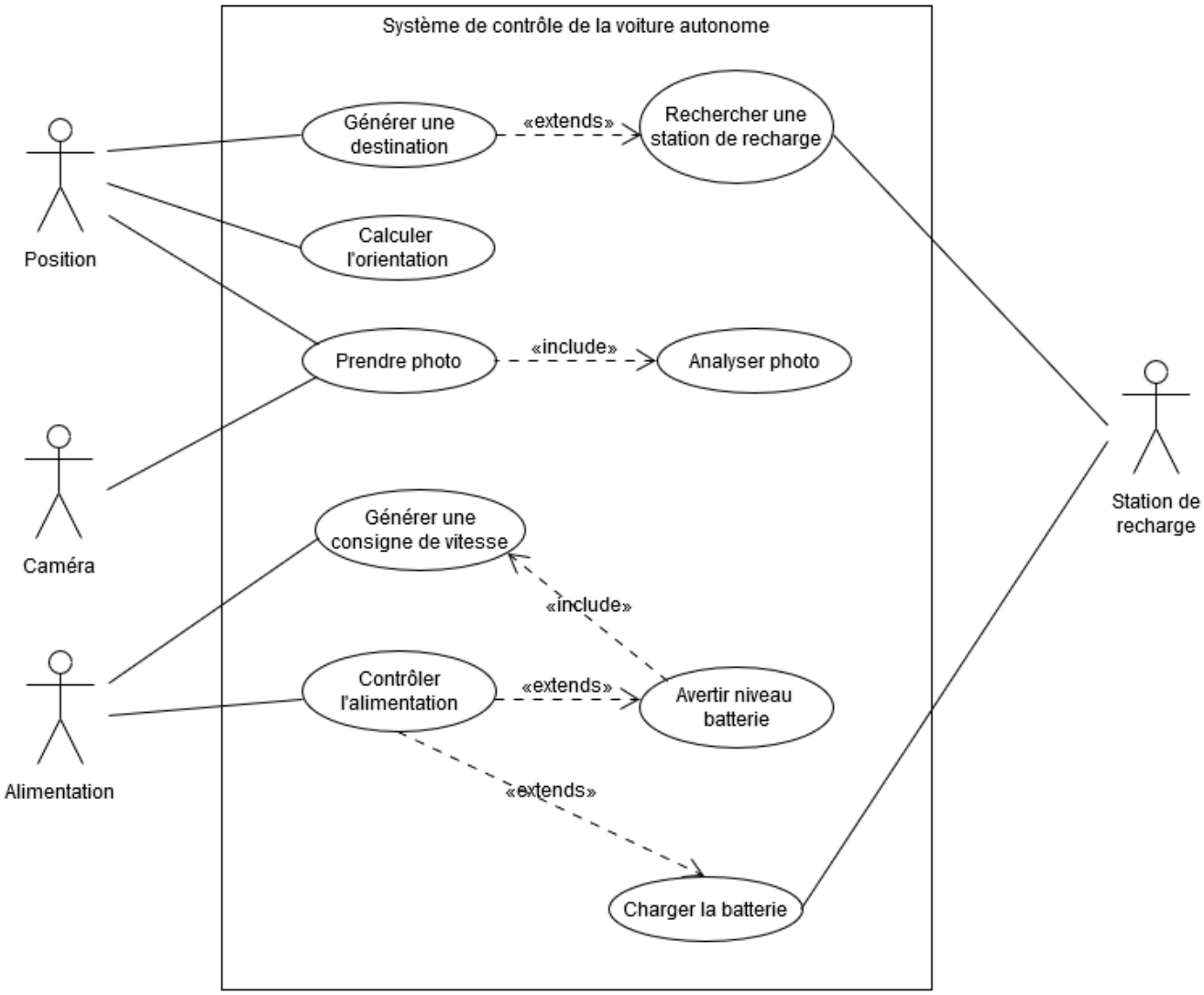



FIGURE 6. Diagramme de cas d'utilisation

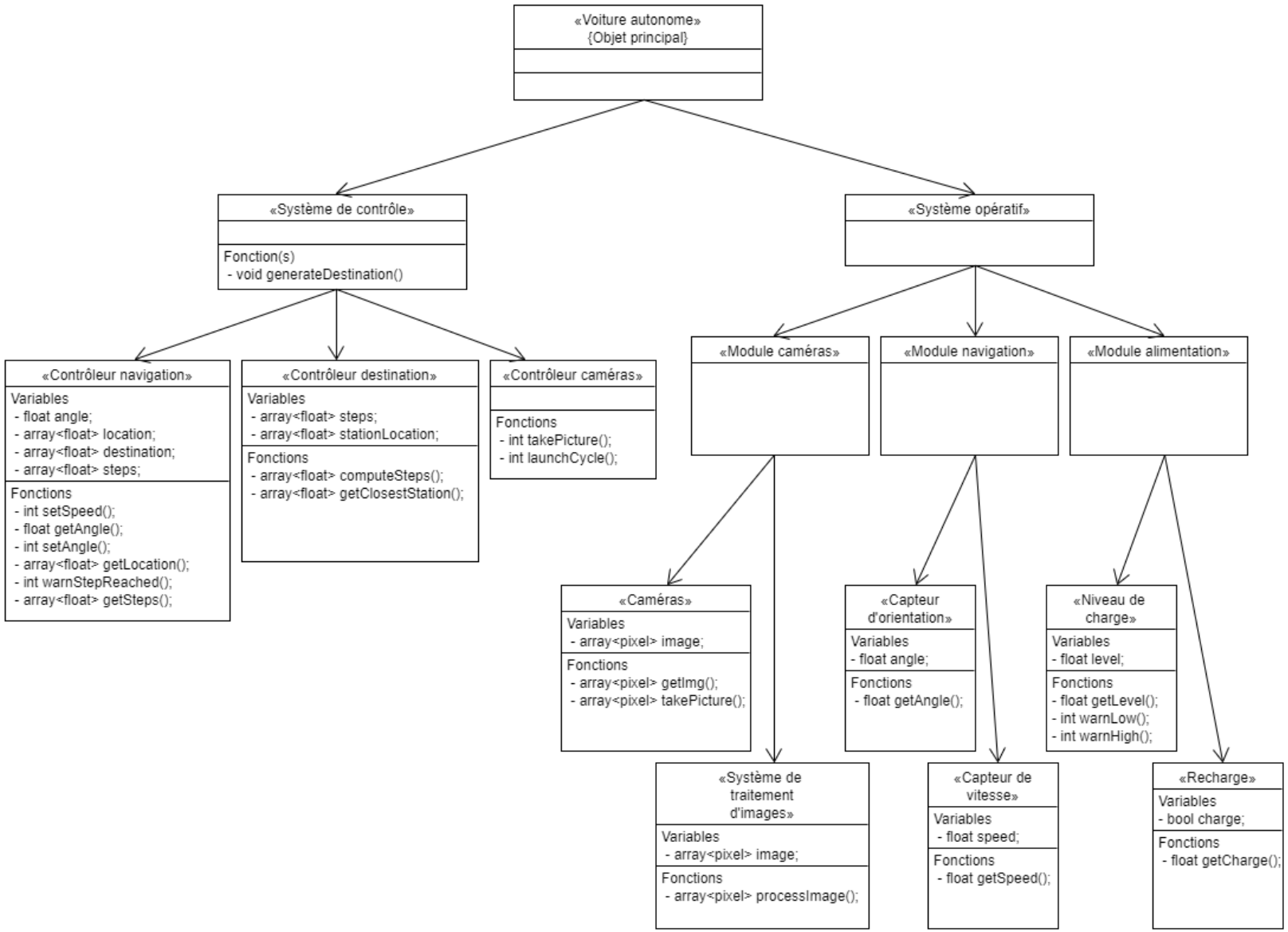
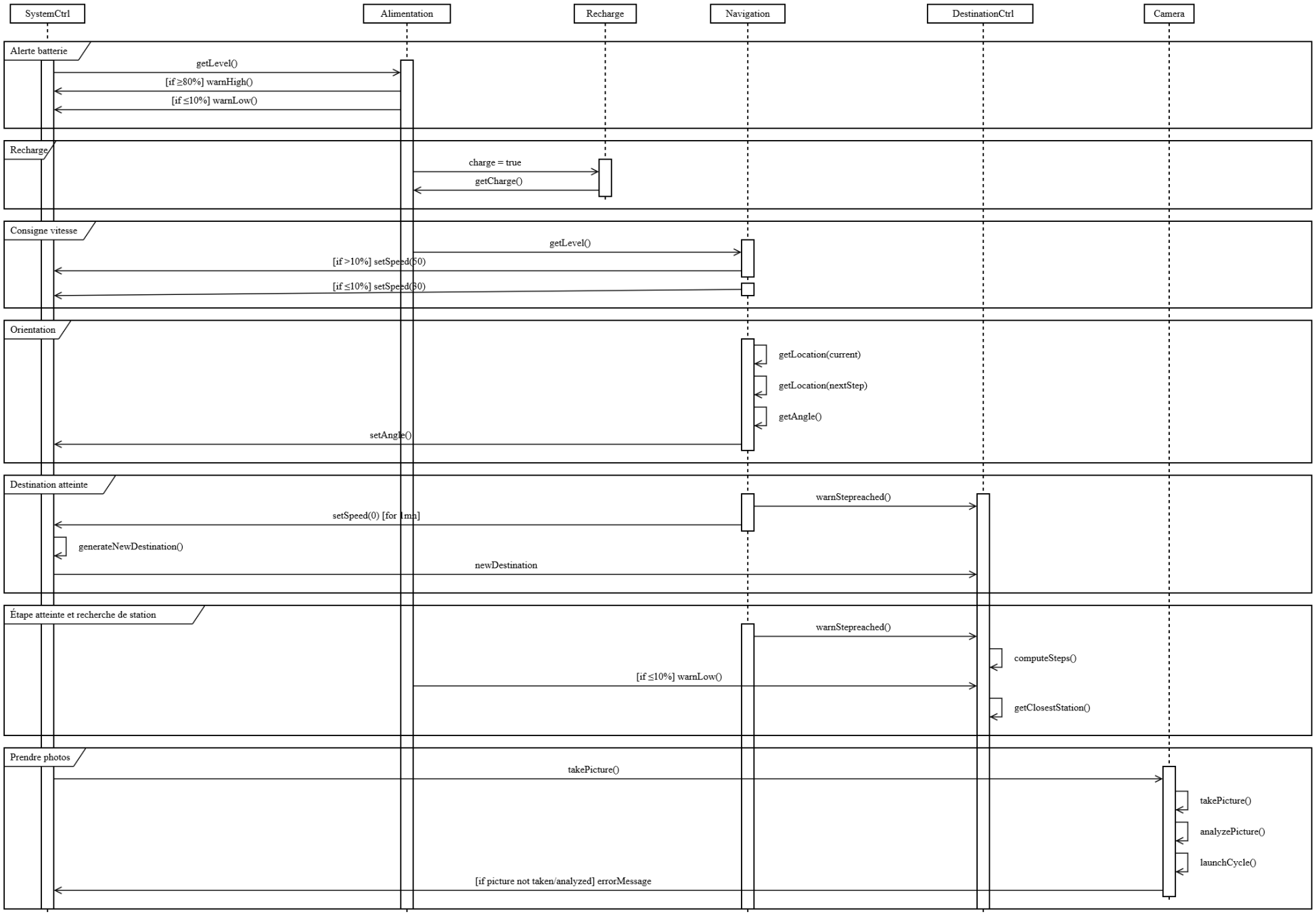



FIGURE 7. Diagramme d'objets


SystemCtrl
Alimentation
Recharge
Navigation
DestinationCtrl
Camera
Alerte batterie
getLevel()
[if ≥80%] warnHigh()
[if ≤10%] warnLow()
Recharge
charge = true
getCharge()
Consigne vitesse
getLevel()
[if >10%] setSpeed(50)
[if ≤10%] setSpeed(30)
Orientation
getLocation(current)
getLocation(nextStep)
getAngle()
setAngle()
Destination atteinte
warnStepreached()
setSpeed(0) [for 1mn]
generateNewDestination()
newDestination
Étape atteinte et recherche de station
warnStepreached()
computeSteps()
[if ≤10%] warnLow()
getClosestStation()
Prendre photos
takePicture()
takePicture()
analyzePicture()
launchCycle()
[if picture not taken/analyzed] errorMessage

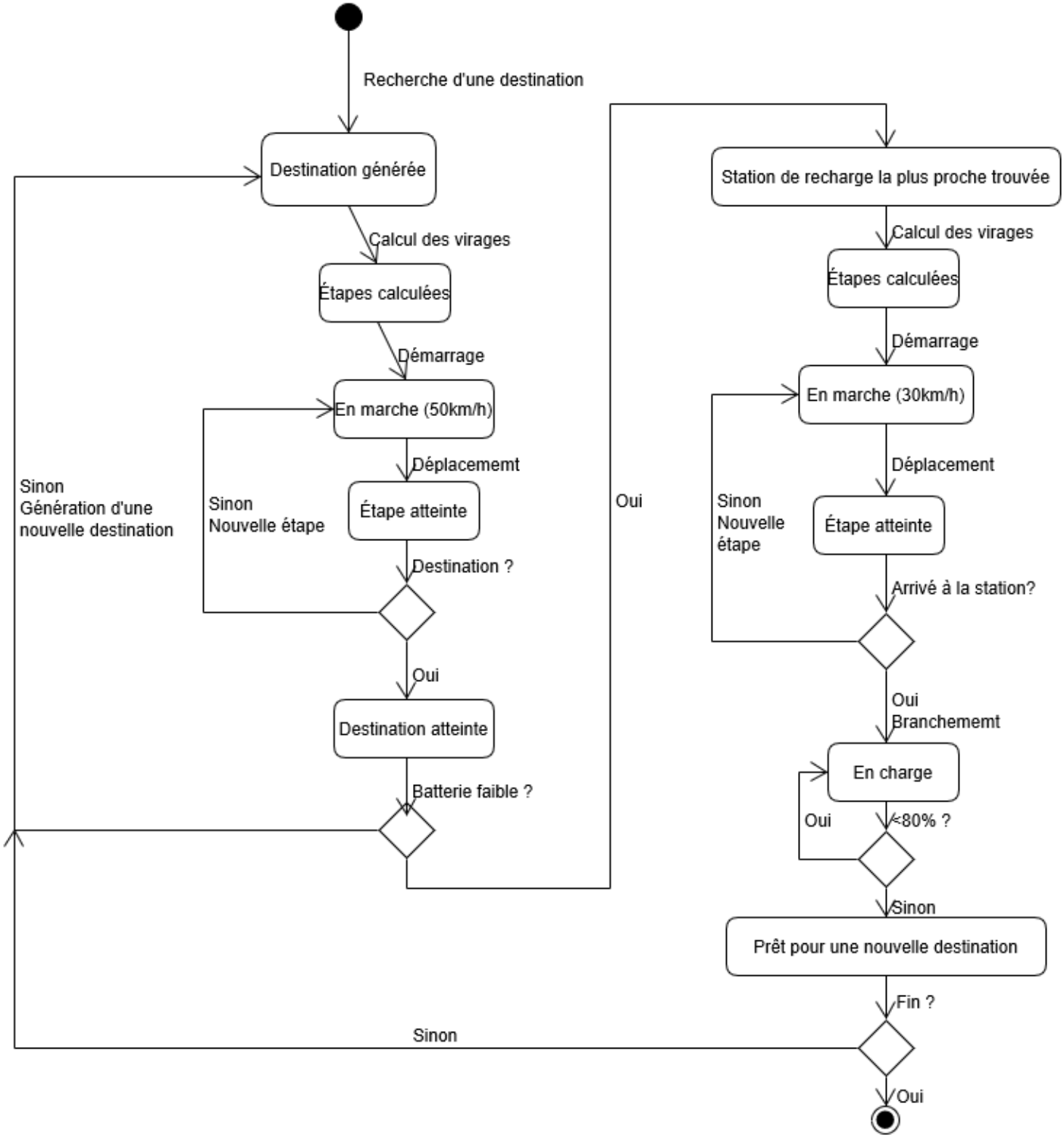



FIGURE 9. Diagramme d'activitité