LA GÉNÉRATION AUTOMATIQUE DE DESCRIPTIONS DE SYSTÈMES DYNAMIQUES

AUTOMATIC GENERATION OF DESCRIPTIONS OF DYNAMIC SYSTEMS

Nicole Tourigny
Guy Lapalme
Département d'informatique et de recherche opérationnelle,
Université de Montréal,
C.P. 6128, Montréal (Québec), Canada H3C 3J7
e-mail:{tourigny,lapalme}@iro.umontreal.ca

Résumé

L'un des problèmes concrets motivant notre recherche est celui associé à la difficulté de comprendre des systèmes dynamiques. Nous avons donc développé un générateur qui en fournit une description en langue naturelle. Le point de départ de la génération est la description du système dynamique exprimée sous forme de spécifications formelles et de traces d'exécution d'un programme de simulation. Nous expliquons notre démarche dans la génération de descriptions de systèmes dynamiques et nous présentons l'état de notre système de génération.

1. Introduction

Parmi les travaux antérieurs de génération de descriptions, plusieurs concernent des objets ou des systèmes d'objets statiques mais peu portent sur les aspects dynamiques d'un système. Cependant, il existe de nombreuses situations où on pourrait tirer avantage d'un système qui générerait automatiquement une description d'un système dynamique. En plus de descriptions générales de systèmes, pensons à la compréhension des programmes, à la description d'un programme de simulation et à la description des processus en cours dans le domaine des télécommunications. Précisons tout de suite que, dans le contexte du présent travail, un système sera dit dynamique ou en évolution, s'il est composé de différents objets pouvant éventuellement interagir entre eux et où les objets peuvent posséder des caractéristiques sous forme d'attributs et de comportements. Il ne s'agit pas ici de systèmes statiques dont les objets peuvent simplement changer d'état mais bien de systèmes d'objets actifs.

Le but du projet est de contribuer au développement de la génération de texte en tenant compte de l'aspect dynamique d'un système dans la production de sa description. L'un des problèmes concrets choisi comme point de départ est celui associé à la difficulté de comprendre des systèmes dynamiques à partir des spécifications formelles, d'une part à cause de la syntaxe du formalisme utilisé mais surtout à cause des interactions possibles entre les différentes parties du code, lesquelles sont difficiles à imaginer à la lecture du programme. Quant à la trace d'exécution, elle donne certes des informations très précises sur le comportement du système, mais elle est habituellement exprimée en termes d'actions particulières des instances d'objet présentes dans le système et n'offre pas une information résumée en termes de classes d'objets; elle ne donne donc pas une idée générale du système.

Par conséquent, lorsqu'on dispose d'un programme et d'une trace d'exécution, il n'est pas toujours facile d'imaginer le comportement réel du système. Il serait alors avantageux de disposer d'un générateur qui puisse nous en fournir une description générale en langue naturelle. Le problème peut donc être reformulé de la façon suivante:

> Est-il possible de décrire en français un système dynamique en ne considérant que ses spécifications formelles et une trace d'exécution du programme? Et, si c'est possible, le résumé produit sera-t-il satisfaisant, c'est-à-dire reflétant le système et compréhensible? Sinon, quels sont les ajouts à faire pour que la description soit acceptable?

Dans ce contexte, les sous-problèmes à résoudre directement reliés à l'aspect dynamique de la génération sont les suivants:

- identification des informations nécessaires en termes de classes d'objets et d'interactions de classes d'objets présents dans le système;
- localisation de ces informations;
- sémantique des informations;
- interprétations du formalisme des spécifications;
- utilisation des informations de la trace;
- identification du but du système.

Par ailleurs, d'autres aspects plus généraux de la génération de descriptions devront être considérés:

- planification et organisation de la description du système;
- aspects syntaxiques et lexicaux;
- utilisation d'une grammaire et d'un lexique;
- justification des choix;
- modes de représentation des connaissances.

Ce projet concerne donc quelques problèmes importants associés à la génération de texte et les différentes théories du discours proposées dans ce domaine doivent être prises en compte. Toutefois, il s'agit ici d'un premier travail dans ce domaine et nous n'espérons pas résoudre d'un seul coup tous ces problèmes. Dans un premier temps, nous nous sommes intéressés plus spécifiquement à la planification de la génération automatique de descriptions de systèmes dynamiques. Le but du présent article est d'expliquer notre démarche et de présenter l'état de notre système de génération, notre première implantation.

2. Présentation du système

Selon Birtwistle (1983), "la simulation est une technique pour représenter un système dynamique au moyen d'un modèle en vue d'obtenir de l'information au sujet du système sous-jacent". Par conséquent, il nous apparaît raisonnable de mener nos expériences de génération de descriptions de systèmes dynamiques en utilisant un langage de simulation.

Le point de départ de la génération sera donc la description du système dynamique exprimée sous forme de spécifications formelles et de traces d'exécution du programme de simulation. Les spécifications formelles permettront de décrire individuellement de façon statique les différentes entités du système au niveau de leur structure et de leur comportement. La trace d'exécution de ce programme donnera de l'information au sujet des interactions entre les entités. A partir de ces données, une description textuelle générale du système décrit par ce programme sera générée. Soulignons que l'originalité de notre projet réside dans cette prise en compte des éléments de la trace et de l'intégration de ces informations à celles reliées aux spécifications formelles en vue de produire un résumé général du système étudié. Notre contribution se situe donc au niveau d'une réflexion et d'une réalisation au niveau de la prise en compte des aspects dynamiques dans la génération de descriptions générales d'un système.

Le langage choisi est le langage de simulation DEMOS (Birtwistle 1983), une extension du langage Simula. DEMOS permet de décrire de façon formelle des systèmes dynamiques et fournit un mécanisme pour la trace d'exécution, un instrument très important dans le contexte de la tâche que nous nous proposons.

Un exemple adapté de Birtwistle (1983) permettra de mieux fixer les idées. La figure 1 donne une description visuelle de la situation: un traversier transportant des autos entre la terre ferme et une île. Les figures 2 et 3 montrent respectivement le programme DEMOS correspondant et une trace partielle d'exécution de ce programme. Le projet consiste à définir un générateur capable de produire une description en langue naturelle de la situation à partir des spécifications formelles et de la trace d'exécution.

A partir du code DEMOS correspondant à l'exemple donné, il est facile d'obtenir la liste des entités présentes et la liste des activités prévues. Cependant, les spécifications formelles ne fournissent pas toujours une information complète. Par contre, la trace d'exécution énumère tous les événements survenus durant le processus et permet donc d'identifier facilement les interactions réelles entre les objets et, par conséquent, les classes interagissant entre elles. Dans notre exemple, les questions qui surgissent sont:

- Quelle entité attend une auto ? (instruction cq(1).wait à la ligne 8, figure 2)
- Quelle entité attend le traversier ? (instruction E:-cq(Cote).coopt à la ligne 22 de la figure 2)

L'information pourrait être déduite des spécifications. Cependant, il est plus simple de consulter la trace. D'un point de vue global, le problème peut être scindé de la façon suivante: recherche d'information, création du message interne et génération de ce message en langue naturelle. Les principaux points à prendre en compte lors de la recherche d'information concernent l'identification des entités en interaction, la délimitation des parties de code partagées par ces entités lors de la synchronisation ainsi que l'instanciation des variables apparaissant dans les instructions impliquées. Dans notre exemple, nous avons d'abord repéré dans les spécifications du programme (figure 2 pour la version DEMOS et figure 6 pour la version Prolog) la liste des entités, leurs actions respectives et les composantes de chacune de ces dernières en termes d'activités ultimes ou exécutables. Lorsqu'une activité correspond à une instruction de synchronisation, nous consultons la trace pour identifier les entités impliquées ainsi que les activités à ajouter à la liste initiale. Cet ajout est réalisé en instanciant au passage les variables apparaissant dans les instructions copiées. Par ailleurs, il faut également instancier les variables correspondantes contenues dans la liste des activités de la seconde entité. Ayant trouvé l'information désirée, les messages internes sont construits pour chacun des cas. Puis, nous avons écrit une composante de génération de surface permettant de produire un texte français à partir de ces formats internes. Voici un exemple de description produite par notre générateur pour la situation décrite par les figures 1, 2 et 3:

```
Il y a 2 entités: auto et traversier.

les actions d'une auto sont arriver,
livrer et retourner tandis que les
actions d'un traversier sont attendre,
charger, traverser, décharger et
continuer.

une auto arrive au quai principal où
elle attend un traversier. ses activités
sont ensuite partagées avec cette
dernière entité: entrée dans la file,
chargement, traversée, déchargement,
placement en tête de la file et sortie
de la file. puis, elle reprend ses
activités propres. elle livre: elle fait
le tour de l'île. ensuite, elle retourne
au quai de l'île où elle attend un
traversier.
```

```
un traversier attend: il fait une file
d'attente. il charge: il dirige une auto
au quai principal, il entre une auto
dans la file et il fait le chargement.
il traverse: il fait la traversée. il
decharge: il fait le déchargement, il
place une auto à la tête de la file, il
sort une auto de la file et il active
l'auto. il continue: il repete ses
activités.
```

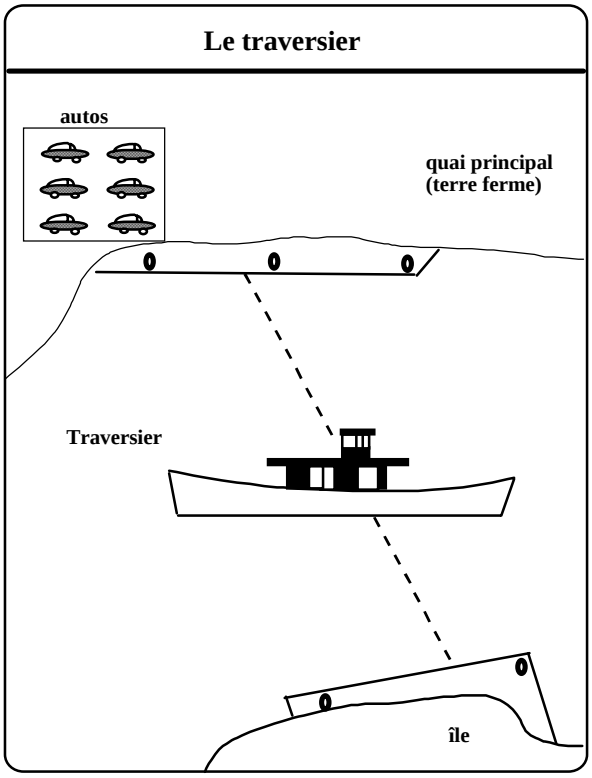

figure 1: description visuelle schématisée du système dynamique

Cette description montre l'intégration des informations. En effet, le troisième paragraphe nous apprend qu'une auto attend un traversier; sont ensuite explicitées les activités subséquentes de l'auto, lesquelles n'apparaissaient dans les spécifications du programme que dans le code du traversier; finalement, toutes les variables ont été correctement instanciées avec les valeurs "auto" ou "traversier" selon le cas, assurant les bons compléments dans le texte.

Dans les prochaines sections, nous expliciterons le modèle et le système ayant servi à produire la description ci-dessus.

3.Un modèle de descriptions de systèmes dynamiques

Selon McKeown (1985), les problèmes fondamentaux de la génération de texte peuvent se résumer à décider quoi dire, dans quel ordre dire le message et comment le dire. Lorsqu'on veut générer un texte, il faut donc, avant d'en générer la forme finale, le planifier, soit décider de son contenu et de son organisation. Nous présentons ici le modèle servant à élaborer le message dans la génération de descriptions de systèmes dynamiques. Nous le situerons ensuite par rapport aux travaux antérieurs.

```
1  COMMENT références: Birtwistle, p. 77 - Le Traversier ;
2  BEGIN EXTERNAL CLASS DEMOS="demos2";
3  DEMOS
4     BEGIN
5       REF(RDIST)CHARGEMENT,DECHARGEMENT,TRAVERSEE,TOURILE;
6       REF(WAITQ)ARRAY CQ(1:2);

7       ENTITY CLASS  AUTO ;
8       BEGIN     ARRIVER :          CQ(1).WAIT;
9                 LIVRER :           HOLD(TOURILE.SAMPLE);
10                RETOURNER :        CQ(2).WAIT;
11    END****AUTO****;

12    ENTITY CLASS  TRAVERSIER ;
13    BEGIN INTEGER C,COTE;
14        REF(QUEUE)CARGO; REF(ENTITY)E;
15              CREER_QUEUE :  CARGO:- NEW QUEUE("CARGO");
16          LOOP:
17              FOR COTE:= 1,2 DO
18              BEGIN
19                CHARGER :        C:= 0;
20                                 WHILE C<6 AND CQ(COTE).LENGTH >0 DO
21                                 BEGIN
22                                   E:- CQ(COTE).COOPT; E.INTO(CARGO);
23                                   HOLD(CHARGEMENT.SAMPLE);  C:= C+1;
24                                 END;
25                TRAVERSER :      HOLD(TRAVERSEE.SAMPLE);
26                DECHARGER :      WHILE CARGO.LENGTH > 0 DO
27                                 BEGIN
28                                   HOLD(DECHARGEMENT.SAMPLE);
29                                   E:- CARGO.FISRT;   E.OUT; E.SCHEDULE(0.0);
30                                 END;
31              END;
32                CONTINUER:              REPEAT;
33    END****TRAVERSIER****;

34  COMMENT programme principal;

35  TRACE;
36  CQ(1):- NEW WAITQ("PRINCIPAL");
37  CQ(2):- NEW WAITQ("ILE");
38  CHARGEMENT:-NEW NORMAL("CHARGEMENT",0.01,0.0);
39  DECHARGEMENT:- NEW NORMAL("DECHARGEMENT",0.01,0.0);
40  TRAVERSEE:- NEW NORMAL( "TRAVERSEE",1.0,0.0);
41  TOURILE:- NEW NORMAL("TOURILE",1.0,0.0);
42  NEW TRAVERSIER("TRAVERSIER").SCHEDULE(0.0);
43  NEW AUTO("AUTO").SCHEDULE(0.0); NEW AUTO("AUTO").SCHEDULE(0.0);
44  NEW AUTO("AUTO").SCHEDULE(0.0); NEW AUTO("AUTO").SCHEDULE(0.0);
45  NEW AUTO("AUTO").SCHEDULE(0.0); NEW AUTO("AUTO").SCHEDULE(0.0);

46  HOLD(10.0);
47      END;
48  END;
```

figure 2: programme DEMOS

```
...
    time/ current     and its action(s)

    0.000 mpdem       schedules TRAVERSIER 1
now
                  schedules AUTO 1 now
                  schedules AUTO 2 now
                  schedules AUTO 3 now
                  schedules AUTO 4 now
                  schedules AUTO 5 now
                  schedules AUTO 6 now
                  holds for 10.000, until 10.000
        TRAVERSIER 1 holds for 1.000, until
1.000
        AUTO 1      waits in PRINCIPAL
        AUTO 2      waits in PRINCIPAL
        AUTO 3      waits in PRINCIPAL
        AUTO 4      waits in PRINCIPAL
        AUTO 5      waits in PRINCIPAL
        AUTO 6      waits in PRINCIPAL
    1.000 TRAVERSIER 1 holds for 1.000, until
2.000
    2.000           coopts AUTO 1 from
PRINCIPAL
                  holds for 0.010, until 2.010
    2.010           coopts AUTO 2 from
PRINCIPAL
                  holds for 0.010, until 2.020
    2.020           coopts AUTO 3 from
PRINCIPAL
                  holds for 0.010, until 2.030
    2.030           coopts AUTO 4 from
PRINCIAL
                  holds for 0.010, until 2.040
    2.040           coopts AUTO 5 from
PRINCIPAL
                  holds for 0.010, until 2.050
...
```

figure 3: trace partielle d'exécution DEMOS

On peut considérer un système en évolution comme étant constitué d'un ensemble d'objets interagissant entre eux en vue d'accomplir un but. Chaque objet est décrit de façon statique dans les spécifications du programme à l'aide d'attributs et de comportements (aussi appelés procédures). Par ailleurs, l'exécution du programme permet de connaître le comportement global du système et c'est la trace d'exécution qui nous fournit l'information; c'est ce qu'on appelle l'aspect dynamique du système. On suppose donc qu'un système peut être décrit en considérant l'information venant de ses aspects statiques et dynamiques et en intégrant le tout (figure 4). La question qu'on se pose ici est: comment choisir et organiser l'information en vue de décrire de façon générale et satisfaisante le système à partir de ses aspects statiques et dynamiques. Soulignons ici qu'un programme de simulation ne représente toujours qu'un modèle du système réel, ce qui signifie une simplification plus ou moins grande de la réalité. Il peut donc arriver que l'absence de certains détails entraîne une description incomplète, voire inadéquate.

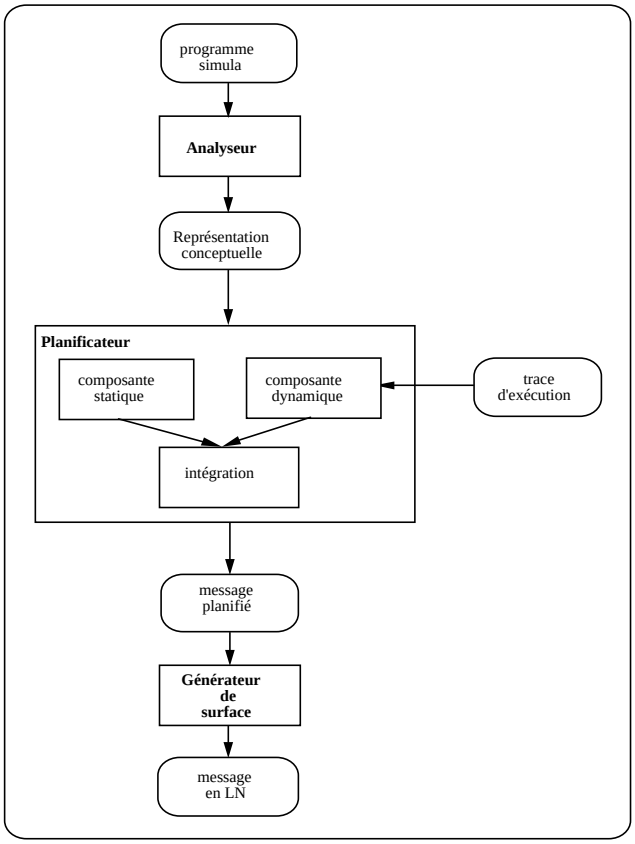

figure 4: Vue générale du système de génération de descriptions dynamiques

3.1 Aspects statiques

A partir des spécifications formelles on peut articuler le système en effectuant les opérations suivantes:

- identification des objets présents dans le système; selon le langage de simulation choisi, ces objets peuvent prendre différentes formes; en DEMOS, ce sont des ressources, des entités, etc.;
- identification des attributs et des comportements pour chaque objet (aussi appelées procédures, actions, activités);
- pour chaque caractéristique, attribut ou comportement, identification des sous-items associés.

Ces opérations peuvent également être réalisées par le planificateur de descriptions. De plus, la notion de schéma permet de les ordonnancer afin de construire un message interne. Par exemple, on pourrait définir le schéma général suivant pour la description :

résumé --> introduction,
développement,
conclusion

chacune de ces parties étant à son tour définie. Ainsi, on aura:

introduction --> informations générales

développement --> pour chaque objet,
énumérer les attributs et actions
énumérer les constituants
informations générales --> nommer les types d'objets
nommer les types d'objets --> identification des ressources,
identification des entités

Il est donc possible de définir des schémas et des opérations (ou prédicats) pour retrouver l'information permettant de décrire les aspects statiques d'un système à partir des spécifications formelles exprimées dans le programme. La section ultérieure portant sur l'implantation permettra d'illustrer cette notion.

3.2 Aspects dynamiques

Les caractéristiques (attributs ou procédures) d'un objet peuvent être exclusives à cet objet, être héritées d'un ancêtre ou être communes à d'autres objets. Les spécifications du programme permettent de retrouver une caractéristique explicitement nommée dans la définition d'un objet ou héritée et cet aspect a été traité ci-dessus, dans les aspects statiques. Cependant, il arrive que le comportement d'un objet comporte des opérations qui ne sont pas explicitement nommées dans cet objet ou dans les ancêtres. C'est le cas des activités de synchronisation dans un programme, par exemple. La présence de certaines instructions dans la définition d'un comportement indiquera qu'il peut y avoir des parties de comportements explicitement nommées dans un autre objet. L'analyse de la trace pourra aider à identifier les caractéristiques communes à certains objets. Dans ce cas, on devra repérer l'objet interagissant et voir s'il faut ajouter explicitement les actions dans la liste des actions des objets concernés.

Ces instructions varient selon le langage choisi mais elles doivent pouvoir se regrouper en catégories. Ainsi, en DEMOS, selon les buts de modélisation suivis, diverses instructions peuvent être utilisées:

- la coopération entité-entité permet de modéliser des entités conservant un comportement propre dans le système mais pouvant jouer des rôles dits actifs (on parle alors d'entité *maître*) ou passifs (on parle d'entité *esclave*) à certains moments; ce type de synchronisation est réalisé à l'aide des instructions wait, coopt et schedule; l'exemple du système représentant des autos attendant un traversier et dont on veut que chaque entité garde un comportement propre illustre ce type de synchronisation; c'est l'exemple que nous verrons en détail un peu plus loin;

- la synchronisation entre une entité et des ressources peut être réalisée de deux façons: l'*exclusion mutuelle* et le modèle de *production/consommation*; dans l'exclusion mutuelle, les ressources requises par une entité sont relâchées par la même au moyen des instructions acquire et release; par exemple, une entité *bateau* peut requérir une ressource *remorqueur* pour entrer dans un port (acquire) et la libérer après utilisation (release). Dans la seconde façon, une entité productrice rend les ressources disponibles à une entité consommatrice et les instructions concernées sont give et take; ce mécanisme de synchronisation peut permettre de représenter une file d'autos attendant puis prenant un traversier; une classe *arrivée* peut gérer l'ajout des autos dans la file (give) et une autre classe *traversier* celui des retraits (take); dans notre exemple, nous n'avons pas d'occurrence de ce modèle de synchronisation;

- la modélisation d'entités devant attendre que toutes les ressources requises soient disponibles avant de démarrer une activité est réalisée au moyen des instructions waituntil et signal; c'est le cas d'un bateau qui doit utiliser deux remorqueurs et un appontement et attendre que la marée ne soit pas basse pour accoster; la première instruction signifie qu'il attend la disponibilité de toutes les ressources et la seconde signale au système la libération de ressources, ce qui les rend à nouveau disponibles.

La modélisation de systèmes dynamiques en DEMOS fait appel à ces catégories d'instructions. Pour modéliser le processus permettant de passer des spécifications formelles d'un système dynamique à sa description en langue naturelle en passant par la trace, il faut donc pouvoir analyser chacune des ces catégories. Cependant, faute d'espace, nous nous limiterons dans cet article à la *coopération entité-entité* car elle permet de modéliser simplement le cas qui nous apparaît le plus fondamental, soit celui des systèmes comportant des entités autonomes en interaction.

3.2.1 Coopération entre les entités

Dans ce type de synchronisation, on veut modéliser le comportement des objets comme des entités ayant un comportement propre et interagissant avec d'autres entités. Les instructions wait, coopt et schedule sont alors utilisées. Pour illustrer le fonctionnement, reprenons l'exemple d'une auto qui prend un traversier. On modélise l'auto et le traversier comme deux entités, chacune des entités ayant son propre comportement. Cependant, lorsque l'auto est sur le traversier, son comportement dépend de celui du traversier; l'auto est alors une entité passive tandis que le traversier est l'entité active.

Les définitions des entités *auto* et *traversier* sont définies respectivement par les classes AUTO et TRAVERSIER (figure 2, lignes 7 à 11 et lignes 12 à 33).

Selon la définition de la classe AUTO, l'activité ARRIVER (ligne 8) consiste à faire un appel wait à CQ(1). Selon la sémantique de DEMOS, CQ(1) est un objet de

type waitq comportant 2 files: slaveq retient les victimes potentielles et masterq les maîtres potentiels si slaveq est vide. Par l'instruction CQ(1).wait, l'auto signale alors qu'elle est prête pour la traversée; puis elle se met à dormir dans la file d'attente slaveq de CQ(1), ce qui réveille le traversier si ce dernier attendait l'arrivée d'une auto dans la file d'attente masterq de CQ(1). A partir de ce moment, l'auto attend passivement jusqu'à ce que le traversier la dépose sur l'autre rive. Elle reprend alors le cours de ses actions propres à l'étiquette LIVRER (ligne 9) lorsque réactivée comme entité autonome par le traversier (instruction schedule, ligne 29) après la traversée.

Quant au traversier, il attend un client sur la rive principale en faisant CQ(1).coopt (ligne 22), ce qui le met dans la file masterq de CQ(1) si la file slaveq de CQ(1) est vide. Un appel à coopt diffère l'action de l'émetteur jusqu'à ce qu'il y ait une entité passive et retourne une référence à la première entité dans la file slaveq s'il y en a une; il enlève aussi l'esclave de cette file. L'entité auto demeure donc passive pendant les trois instructions hold suivantes (lignes 23 à 28, dans la description de traversier); tel que mentionné ci-dessus, elle reprend ses activités à l'étiquette LIVRER après que le traversier ait fait E.schedule(0.0) à la ligne 29, ce qui permet à l'auto de redevenir une entité ayant son comportement propre à l'heure courante (now ou 0.0).

Lorsqu'on lit le programme, on peut se poser les questions suivantes:

- Lors de l'appel CQ(1).wait par AUTO, que représente CQ(1)? Le code dit qu'il s'agit d'un élément d'un tableau, identifié par l'étiquette "principal" et représentant une file (type waitq). Mais que représente-t-il de la réalité simulée?
- Lorsque le traversier émet l'appel CQ(COTE).coopt, quelle entité attend-il?

Par ailleurs, dans la trace, dont le format en Prolog[1] (figure 5) est

```
trace(heure, entité,noDinstance, action)
```

on peut lire aux lignes 1 et 3:

```
trace(0.000,auto,1,wait(principal)).
trace(2.000,traversier,1,coopt(auto,1,
            principal)).
```

On a donc une instance de AUTO qui lance un wait à un certain objet et une instance de TRAVERSIER qui fait un coopt de la première instance pour le même objet. On peut analyser la trace en cherchant des formats particuliers. Pour l'exemple précédent, la recherche des prédicats

```
trace(T1,Entite1,N,wait(Lieu))
trace(T2,Entite2,1,coopt(Entite1,N,Lieu)))
```

et l'ajout de la contrainte

```
T2 >= T1
```

```
1  trace(0.000,auto,1,wait(principal)).
2  trace(1.000,traversier,1,hold(1.000,2.000)).
3  trace(2.000,traversier,      1,coopt(auto    ,1principal)).
4  trace(2.000,traversier,      1,hold(0.010,2.010)).
5  trace(2.010,traversier       ,1,hold(1.000,3.010)).
6  trace(3.010,traversier       ,1,hold(0.010,3.020)).
7  trace(3.020,traversier       ,1,schedule(auto      ,1,now)).
8  trace(3.020,ferry,1,hold(1.000,4.020)).
9  trace(3.020,traversier       ,1,hold(1.000,4.020)).
10 trace(4.020,traversier       ,1,hold(1.000,5.020)).
11 trace(4.020,auto,1,wait(ile)).
12 trace(5.020,traversier       ,1,coopt(auto    ,1,ile)).
13 trace(5.020,traversier       ,1,hold(0.010,5.030)).
14 trace(5.030,traversier,      1,hold(1.000,6.030)).
15 trace(6.030,traversier,1,hold(0.010,6.040)).
16 trace(6.040,traversier,1,schedule(auto              ,1,now)).
17 trace(6.040,traversier,1,hold(1.000,7.040)).
18 trace(6.040,auto      ,1,terminates).
19 trace(7.040,traversier,1,hold(1.000,8.040)).
20 trace(8.040,traversier,1,hold(1.000,9.040)).
21 trace(9.040,traversier,1,hold(1.000,10.040)).
   .....
```

figure 5: trace exprimée en Prolog

permettront d'identifier l'entité qu'attend une auto ainsi que l'endroit où elle l'attend. Notons que ces deux prédicats trace n'ont pas à être consécutifs dans le fichier de la trace. Prolog retrouvera dans le fichier les différentes assertions correspondant à cette requête. Ayant déterminé ainsi les entités en interaction pour cet appel wait, on pourra construire la liste des actions communes aux deux entités en ajoutant celles comprises, dans l'entité active, entre les instructions coopt et schedule aux activités de l'entité passive, juste après l'appel wait.

```
1   % définition des classes et objets
2   % classe(superclasse, liste des variables, liste des  procédures, liste des
3   actions, particularités).

4   auto  (entity,[],[],[arriver,livrer,retourner],[]).
5   traversier(entity,[],[],[creer_queue,charger,traverser,decharger,
6                continuer],[]).
7   demos(entity,[],[hold],[],[]).

8   %  définition des actions
9   %  action(classe d'appartenance, liste des paramètres, type d'action,
10              liste des sous-actions).

11  arriver(auto    ,[],[], [wait(cq(1))],[]).
12  livrer(auto   ,[],[],[hold(tourile)],[]).
13  retourner(auto    ,[],[], [wait(cq(2))],[]).

14  creer_queue(traversier,[],[],[new(cargo)],[]).
15  charger(traversier,[], [],[coopt(cq(Cote),E),into(E,cargo),
16                        hold(chargement)],[]).
17  traverser(traversier,[],[],[hold(traversee)],[]).
18  decharger(traversier,[],[],[hold(dechargement),first(E,cargo),
19                        out(E,cargo), schedule(E,now)],[]).
20  continuer(traversier,[],[],[repeat],[]).
```

figure 6: description partielle du programme en Prolog

En résumé, les spécifications du programme servent de point de départ pour générer le message interne. Si un comportement comporte certaines instructions, par exemple wait, la trace est consultée pour connaître l'entité interagissant avec l'entité examinée ainsi que les actions qui n'étaient pas explicites dans la première entité. Et cette analyse doit être faite pour toute activité impliquant

[1] Nous avons transformé les éléments de la trace sous forme de prédicats Prolog afin d'en faciliter la manipulation et la recherche par notre système, lui-même écrit en Prolog.

des processus de synchronisation, que ce soit au moyen des groupes d'instructions coopt/schedule/ wait, give/take, acquire/release ou waituntil/ signal. Pour des raisons évidentes de clarté, nous avons présenté un exemple simple. Cependant, notre générateur doit traiter des situations plus complexes: qu'arrive-t-il lorsque plus de deux entités agissent dans le système ou qu'arrive-t-il lorsqu'une entité fait appel à une procédure?

Par ailleurs, d'autres opérations doivent être faites au niveau de la planification. Par exemple, il faut prévoir à partir de quelle entité la génération sera faite: dans l'exemple ci-dessus la description devrait-elle être générée du point de vue de l'auto, de celui du traversier ou de celui du système général? Dans quel ordre les entités doivent-elles êtres décrites et selon quels critères? De plus, il faut être capable de gérer différents types de répétition; notre générateur en traite quelques-uns, dont la répétition du sujet et celle d'une action déjà décrite, mais il reste d'autres cas non traités.

3.3 Travaux antérieurs

McKeown (1985a; 1985b; 1987) étudie les problèmes de décider du contenu du message et de son organisation ainsi que ceux associés à la mise en forme finale dans la génération de descriptions attributives de classes d'objets dans le contexte d'interrogation d'une base de données (système TEXT). Dans la génération de descriptions de systèmes dynamiques, nous nous intéressons également à la planification du message; nous utilisons la notion de schéma pour construire le message interne à partir des aspects statiques: l'ensemble des classes d'objets et leur description. Nous nous sommes inspirés de la notion de schéma de McKeown sans toutefois reprendre ses définitions puisqu'elle ne traite pas des aspects dynamiques d'un système.

Dans son système TAYLOR, Paris (Paris 1988; Paris et McKeown 1987) reprend le travail de McKeown dans le contexte de la génération de descriptions à partir d'informations encyclopédiques et lui ajoute le schéma "process". Cette recherche est intéressante pour la génération de descriptions de systèmes dynamiques pour les raisons citées pour le travail de McKeown. Paris se limite également aux aspects statiques des connaissances en ce sens que toute l'information à générer est déjà contenue dans la description encyclopédique initiale. Dans la génération de descriptions de systèmes dynamiques, il faut identifier les liens entre les diverses parties des spécifications formelles en consultant la trace d'exécution du système.

La théorie des structures rhétoriques RST (Mann 1988) concerne l'organisation textuelle. Selon cette théorie, un texte est représenté comme une hiérarchie de morceaux textuels liés entre eux par des relations rhétoriques. Condition, antithèse et justification sont des exemples de telles relations. Dans notre première implantation, nous n'avons pas utilisé les travaux de Mann; nous avons préféré nous en tenir à la notion de schéma, plus simple et suffisante pour nos besoins. Cependant, ultérieurement, nous considérerons cette théorie, en particulier pour générer les liens entre les différentes parties de la description et ainsi assurer sa cohérence.

Reiter (1990) s'intéresse au problème d'informer un individu ("hearer") qu'un objet particulier a certains attributs (système FN-KR). Le problème considéré est celui de générer des descriptions attributives relatives à un objet particulier qui soient adéquates, valides et libres de fausses implications possibles, étant donné un utilisateur donné. Le point commun de cette recherche et de la nôtre est la description d'un objet appartenant à des classes d'objet. Un système dynamique étant composé de classes d'objets, les objets possédant des attributs de leur classe et, éventuellement, d'autres attributs et des états particuliers, il pourrait donc être intéressant de pouvoir appliquer le travail de Reiter au niveau de la génération de descriptions. Cependant, nous n'avons pas encore examiné ces liens. Par ailleurs, Reiter ne s'intéresse pas aux connaissances dynamiques d'un système.

Swartout (1983) s'est intéressé à la description en anglais des spécifications formelles d'un programme. Au paraphraseur qui ne pouvait traiter que des aspects statiques des spécifications, Swartout ajouta un explicateur de trace. Ce dernier analyse la trace, relève les éléments intéressants et à l'aide d'un certain nombre d'heuristiques, résume et reformule au besoin les preuves. Un générateur est ensuite utilisé pour produire la description en anglais. Ce travail est intéressant pour la génération de descriptions de systèmes dynamiques car il utilise la trace pour générer une description. Cependant, le système de Swartout produit des informations au niveau des instances alors que nous voulons générer un résumé au niveau des classes d'objets et de leurs interactions. De plus, l'évaluateur symbolique utilisé par Swartout produit beaucoup plus d'informations que le traceur de DEMOS. Les points de départ sont donc différents.

4. Implantation

Dans la présente section, nous décrivons l'état actuel de l'implantation du générateur. Nous décrivons d'abord le système d'un point de vue général, puis nous donnons un aperçu de la génération aux niveaux statique et dynamique et, finalement, l'état du prototype. La figure 4 donne un aperçu général du système. Un analyseur lit le programme et le récrit dans une représentation conceptuelle utilisable (figure 6) par le planificateur, c'est-à-dire sous forme de clauses Prolog. C'est à partir de cette structure que toute information statique pourra être ultérieurement repérée, par exemple la liste des entités, leurs structures, etc. Quant à la trace, elle est produite par DEMOS sous forme de clauses Prolog.

4.1 Génération automatique de descriptions de systèmes dynamiques

De l'information produite par l'analyseur à partir des spécifications du programme, le générateur donne une

description des aspects statiques du système. Nous avons utilisé la notion de schéma pour générer une description. Cette dernière se divise en deux parties: une introduction générale (schéma *introduction*, ligne 1 dans le texte ci-dessous) et une partie principale (schéma *développement*, paragraphe 2 à la fin); elle ne contient pas de conclusion. L'introduction contient les *informations générales* au sujet du système; celles-ci ne concernent que les entités puisqu'il n'y a que ce type d'objets. La partie principale comprend aussi deux parties: l*es actions de chaque entité sont énumérées* (paragraphe 2) et les *constituants* de chacune des activités sont énumérées (paragraphes 3 et 4). Notons que les objets n'ayant pas d'attribut spécifique, seules les actions ont été énumérées pour les entités.

Un premier exemple de résumé produit par notre générateur pour l'exemple donné ci-dessus est le suivant:

```
Il y a 2 entités : auto et traversier.

les actions de l'auto sont arriver,
livrer et retourner et les actions du
traversier sont attendre, charger,
traverser, décharger et continuer.

l'action arriver de l'entité auto
consiste à attendre au quai principal.
livrer consiste à attendre un certain
nombre d'heures. l'action retourner de
auto consiste à attendre au quai de
l'île.

l'action attendre de l'entité traversier
consiste à faire une file. charger
comprend les activités suivantes:
prendre en charge _233 au quai
principal, _233 entre dans la file
d'attente et attendre un certain nombre
d'heures. l'action traverser de l'entité
traversier consiste à attendre un
certain nombre d'heures. décharger
comprend les activités suivantes:
attendre un certain nombre d heures ,
_292 premier de la file, _292 sort de la
file et _292 reprend ses activités
propres. l'action continuer de
traversier consiste à répéter les
activités.
```

Cette description peut être schématisée selon la figure 7. Dans ce texte, les mots de la forme _N, où N est un nombre, représentent des variables non instanciées: on ne connaît pas l'objet à ce moment; ainsi, le traversier prend en charge un objet noté _233, lequel entre dans la file d'attente; par ailleurs, le déchargement implique un objet noté _292. A partir des spécifications formelles du programme, on ne peut faire le lien entre ces deux objets notés _233 et _292. Le générateur communique ici l'information à partir de la description du programme sans prendre en compte la trace. Il permet de mettre en lumière les limites de la génération à partir des spécifications du programme. Dans ce cas, on ne sait pas quelle entité attend l'auto, ni quelle entité attend le traversier. De plus, on ne connaît pas toutes les activités explicites associées à l'entité *auto* ; à cette étape-ci, les seules activités connues de l'entité *auto* sont arriver, livrer et retourner, lesquelles se décomposent respectivement en CQ(1).wait, hold(tourile) et CQ(2).wait, ce que nous appelons les activités ultimes car elles ne se décomposent plus.

Afin d'améliorer la lisibilité du texte, nous avons ensuite exprimé les actions sous la forme active et apporté quelques autres modifications. Le message généré pour l'aspect statique du message devient alors:

```
Il y a 2 entités : auto et traversier.

les actions de l'auto sont arriver,
livrer et retourner et les actions du
traversier sont attendre, charger,
traverser, décharger et continuer.

une auto arrive: elle attend au quai
principal. elle livre : elle fait le
tour de l'île. elle retourne: elle
attend au quai de l'île.

un traversier attend: il fait une file
d'attente. il charge : il dirige _249 au
quai principal, il entre _249 dans la
file et il fait le chargement. il
traverse: il fait la traversée. il
decharge: il fait le déchargement, il
place _308 à la tête de la file, il sort
_308 de la file et il active _308. il
continue: il repete ses activités .
```

On doit donc modifier le générateur pour prendre en compte les informations contenues dans la trace d'exécution afin d'améliorer cette description et c'est ce que nous présentons ci-après.

4.2 Génération tenant compte des aspects dynamiques

On peut enrichir la description ci-dessus en prenant en compte les informations venant de la trace du programme. A partir de celle-ci, il est possible de déterminer les interactions entre les différentes entités et de rendre explicites les activités de chaque entité.

Voyons d'abord les activités ultimes de chaque entité. Le diagramme de la figure 7 illustre les activités de chacune. Ces activités apparaissent à la figure 2 dans la partie de droite: ce sont les instructions exécutables par DEMOS; par ailleurs, les actions ont été écrites à gauche, sous forme d'étiquettes. Les activités ultimes de l'entité *auto* sont:

```
wait(cq(1)); hold(tourile); wait(cq(2));
```

Celles de l'entité *traversier* sont:

```
new(cargo);
coopt(cq(Cote)); into(E,cargo); hold(chargement);
hold(traversee);
hold(dechargement); first(E,cargo); out(E,cargo);
schedule(E,now);
repeat;
```

Dans DEMOS, l'entité *auto* joue le rôle d'"esclave", tandis que *traversier* joue celui de "maître". Lorsque *auto* lance l'appel wait, cette entité attend d'être prise en charge par une entité maîtresse. Cette dernière répond à l'appel en faisant coopt et le termine avec l'instruction schedule. Les activités énumérées entre ces deux appels sont des

activités partagées par les deux entités. C'est ce que nous devons ajouter à l'ensemble des activités nommées dans la description de l'entité "esclave" pour obtenir la liste explicite des activités de l'entité. Par ailleurs, il faut aussi instancier les variables contenues dans les différentes instructions impliquées lors des processus de synchronisation.

Ayant trouvé l'information nécessaire, le message interne est construit et la composante de surface produit la description présentée à la section 2. On remarquera que l'introduction (paragraphe 1) et la description générale des actions (paragraphe 2) sont inchangées, ne contenant pas de variables ni n'instructions ultimes.

Quant aux deux paragraphes suivants, ils se sont nettement améliorés. En effet, en plus de conserver la forme active, certaines phrases ont une structure plus complexe. Ainsi, à la première ligne du troisième paragraphe, on peut lire:

```
une auto arrive au quai principal où
elle attend un traversier.
```

La seconde proposition de cette phrase contient le complément direct identifiant l'entité attendue par l'auto. Ensuite, les activités communes aux deux entités sont identifiées (lignes 3 et 4 de ce paragraphe). On remarquera également que les activités n'ont pas été répétées pour l'action "retourner" de l'entité auto puisqu'elles sont les mêmes que l'action arriver, mis à part le lieu. Quant à la description des activités du traversier (4e paragraphe), elle se distingue de la seconde version statique par l'instanciation des différentes variables: les termes _249 et _308 sont maintenant remplacés par "une auto".

On peut conclure qu'il est possible de décrire automatiquement des systèmes dynamiques en considérant les spécifications formelles et la trace d'exécution. Cependant, on peut voir que ce texte est encore améliorable, ce que nous verrons ci-après.

4.3 État du prototype et travaux ultérieurs

Présentement, le générateur peut produire en français une description des aspects statiques d'un système comportant un nombre quelconque d'objets; cependant, il peut produire une description tenant compte des aspects dynamiques que pour des systèmes d'au plus deux entités. Les instructions de synchronisation qui sont traitées sont celles appartenant à la coopération entre entités: wait, coopt et schedule. Parmi les aspects qu'il reste encore à développer, mentionnons:

- implantation des autres catégories d'instructions de synchronisation;
- ajout d'entités impliquées dans différents processus de synchronisation;
- ajout d'entités faisant appel à des procédures;
- choix du point de vue de la description.

Par ailleurs, d'autres problèmes, ceux-là ouverts, devront être étudiés. La sémantique associée au vocabulaire des spécifications et de la trace est extrêmement importante. En fait, c'est elle qui rend le texte intelligible. Dans l'exemple que nous avons présenté, le système possède de l'information quant aux mots à utiliser dans la description finale; il pourrait, par exemple, trouver le mot "ferry" dans le programme et traduire par traversier à la version finale. Il restera à voir comment on peut améliorer cette sémantique, tant au niveau du contenu que de l'organisation. Voici un autre exemple de problème rencontré et lié à la sémantique dans la réalisation de notre système. Pour indiquer qu'une activité dure un certain temps, l'instruction hold est utilisée. Ainsi, dans l'exemple donné dans cet article, un traversier charge (hold(chargement)), traverse le cours d'eau (hold(traversee)), et décharge (hold(dechargement)). Il est à noter que la sémantique du texte produit est intimement liée à celle des termes de la source, soit les variables au nom significatif (chargement, traversee, dechargement) et les étiquettes charger, traverser et décharger. Si ces mots sont sans rapport avec le modèle simulé, le texte résultant sera incompréhensible, à moins que ne soit associé à ces mots un sens plus juste. Pour générer des textes compréhensibles, il faut donc au moins que le vocabulaire utilisé dans les spécifications formelles et dans la trace soit significatif; si tel n'est pas le cas, il faut ajouter un lexique qui permet de passer des mots des spécifications et de la trace au vocabulaire voulu. En fait, on remarque que l'activité de modélisation de la réalité qui a mené à l'écriture d'un programme de simulation a effectué une opération de simplification sur la réalité; le modélisateur est capable d'évaluer le comportement du système car il a une connaissance supérieure à celle qui est dans le programme. L'activité de génération de descriptions part de la version simplifiée de la réalité, soit les spécifications et la trace, et essaie de reconstituer la réalité. Il est donc tout à fait normal de devoir ajouter des informations au modèle simplifié si on veut tenter de reconstituer fidèlement la réalité.

Par ailleurs, il est possible d'enrichir la description en se servant de différents procédés dont la synonymie. Dans la description que nous avons produite, certaines répétitions sont dues à la traduction quasi littérale des étiquettes et des instructions du programme. A titre d'exemple, citons

```
il traverse : il fait la traversée
```

dans le dernier paragraphe de la description donnée au début de l'article. Vaut-il mieux avoir un texte plus riche ou une description plus fidèle du programme? Tout dépend des objectifs poursuivis: comprendre le programme ou obtenir de jolies descriptions. Un autre exemple de traduction littérale des activités des entités est la suivante. Dans le programme, c'est le traversier qui crée la file d'attente et la gère. La description finale produite reflète ces comportements. Mais, ceci est une modélisation simplifiant la réalité. Encore une fois, est-il

préférable de traduire le programme ou de faire une description plus réaliste?

Un autre problème "ouvert" est celui de l'explicitation des buts: quel est le but ultime du système? Quel est le but de telle action? Cette information, connue du modélisateur ayant encodé le programme, est tout à fait absente des spécifications formelles à moins qu'elle ne soit explicitement décrite dans un commentaire. Par exemple, une auto attend au quai pour traverser le cours d'eau et se rendre à l'île; mais notre système ne peut deviner ces buts. Il ne peut que traduire les différentes actions réalisées en cours de processus. Est-ce à dire qu'il faudrait fournir explicitement cette information au système?

5. Conclusion

Nous avons présenté un générateur de descriptions de systèmes dynamiques à partir des spécifications et de la trace d'exécution du programme. Les spécifications formelles contiennent l'information nécessaire pour décrire les aspects statiques du système et la trace permet d'en connaître les aspects dynamiques, soit les interactions entre ses différents objets actifs de même que la liste explicite de leurs comportements. Nous avons présenté le modèle de génération de descriptions, l'implantation actuelle du système et les travaux à venir. La contribution de notre projet se situe au niveau de la prise en compte des aspects dynamiques dans la description générale d'un système.

La réalisation d'un tel projet permet de poser des problèmes importants impliqués dans la génération de descriptions de systèmes dynamiques tels ceux liés au processus de description générale, à l'identification des connaissances nécessaires, à leur organisation dans un texte et à la sémantique associée au vocabulaire utilisé.

Bibliographie

Birtwistle, G.M. (1983) Discrete Event Modelling on Simula, The Macmillan Press, Hong Kong, 1983 (première impression en 1979).

Mann, W. (1988) "Text Generation: The Problem of Text Structure", Natural Language Generation Systems, D.D. McDonald et L. Bolc (éd), Springer-Verlag, New-York, 1988, pp. 47-67.

McKeown, K.R. (1985a) "Discourse Strategies for Generating Natural-Language Text", Artificial Intelligence, Elsevier Science Publishers, North-Holland, 1985, numéro 27, pp. 1-41.

McKeown, K.R. (1985b) Text Generation, Using discourse strategies and focus constraints to generate natural language text, Cambridge University Press, New York, 1985.

McKeown, K.R. (1987) "Language Generation and Explanation", Annual Review of Computer Science, J.F. Traub, B.J. Grosz, B.W. Lampson et N.J. Nilsson (éd.), volume 2, 1987, Palo Alto (Californie), pp. 401-449.

Paris, C.L. (1988) "Tailoring Object Descriptions to a User's Level of Expertise, Computational Linguistics, volume 14, numéro 3, septembre 1988, pp. 64-78.

Paris, C.L. et Mckeown, K.R. (1987) "Discourse Strategies for Describing Complex Physical Objects", Natural Language Generation, New Results in Artificial Intelligence, Psychology and Linguistics, Gerard Kempen (éd.), Martinus Nijhoff Publishers, Dordrecht, The Netherlands, 1987, pp. 97-115. (Proceedings of the NATO Advanced Research Workshop on "Natural Langage Generation", Nijmegen, The Netherlands, August 19-23, 1986).

Reiter, E. B. (1990) Generating Appropriate Natural Language Object Descriptions, Thèse de doctorat, numéro 9035542, Havard University, 1990.

Swartout, B. (1983) "The GIST Behavior Explainer", AAAI-83, Proceedings of the National Conference on Artificial Intelligence, 22-26 août 1983, Washington.

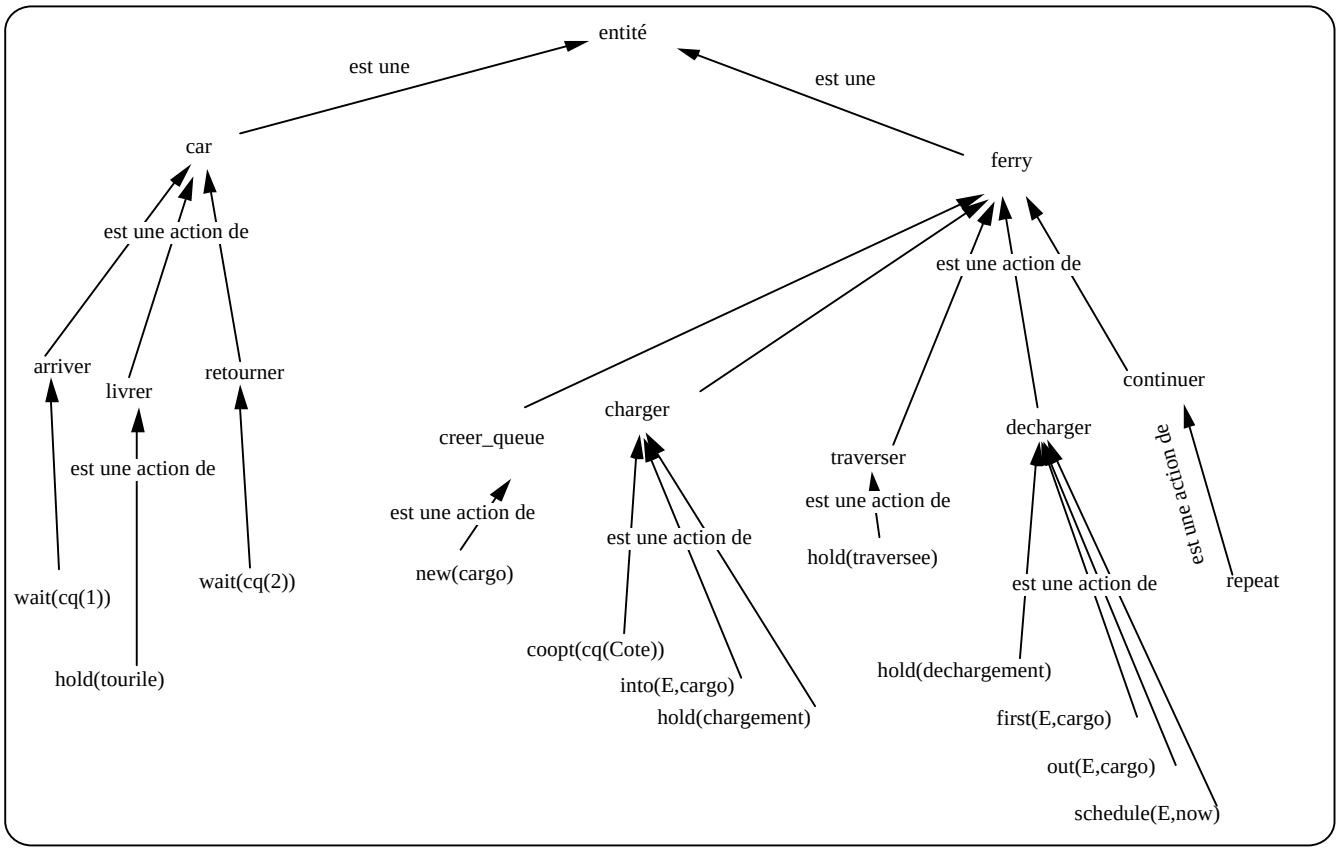

figure 7: activités des entités (selon les spécifications)