Contents

Rapport-gratuit.com
LE NUMERO 1 MONDIAL DU MÉMOIRES

Chapter 1

Introduction

La robotique mobile a révolutionné la réalisation de tâches en environnement hostile : recherche, assistance et exploration ou encore démantèlement ne représentent que quelques exemples parmi les nombreuses missions où on peut de plus en plus éviter de risquer des vies.

L'expérience a toutefois montré les limites d'un aspect de l'approche robotique : l'envoi d'une unité de matériel très sophistiqué (et donc cher) peut ne pas s'avérer pertinent. Le désastre de la centrale Fukushima Daiichi l'a illustré récemment avec la panne définitive, au bout de deux heures seulement, du robot unique chargé d'inspecter et nettoyer une partie du site dévasté, détruit par les radiations[1].

La perte de matériel coûteux[2] et l'avortement de missions importantes sinon critiques (au sens où des vies sont en jeu) invitent à peser les alternatives. Les essaims de robots mobiles s'inscrivent dans cette réflexion.

Dans ce cadre, c'est une flotte de robots qui est à l'œuvre. Ses éléments, *simples*, autonomes, doivent *coopérer* à la réalisation de la tâche considérée, complexe, et se réorganiser face aux évolutions de l'environnement. Une fois déployé, l'essaim peut permettre, en parallélisant les actions et en partageant les risques, l'établissement de réseaux dynamiques, les patrouilles, les explorations de zones de catastrophes, l'assistance aux équipes de secours, etc.

L'emploi d'un essaim de petits robots simples à la place d'un agent unique sophistiqué répond à l'hostilité de l'environnement et en particulier à la quasi-certitude qu'on va perdre des éléments.

D'une part, en utilisant des robots en essaim, plus *simples* et bien meilleur marché qu'un robot spécialisé, l'opérateur, peut donc se permettre d'en perdre.

D'autre part, par le développement de protocoles distribués *robustes*, la tâche elle-même peut supporter la perte de participants : elle n'est plus forcément mise en péril par une défaillance.

On peut ainsi caractériser deux aspects :

- Des capacités faibles pour chaque agent,
- Des propriétés fortes sur le protocole (l'algorithme, unique, embarqué et exécuté par tous les agents),

qui vont s'articuler dans cette approche versatile.

Certains des quelques scénarios que nous avons proposé ici ne tolèrent pas d'échec : des vies peuvent être en jeu. Les solutions à base de flottes de robots qui répondent à des missions critiques doivent donc être soumises à des étapes de vérification intransigeantes.

[1] https://www.theverge.com/2017/2/10/14580674/fukushima-record-high-radiation-cleaning-robot-recalled
[2] https://www.theverge.com/2017/2/17/14652274/fukushima-nuclear-robot-power-plant-radiation-decomission-tepco

Du point de vue informatique, les essaims s'inscrivent dans les thématiques de l'algorithmique distribuée et ils en partagent les difficultés. La certification de ces systèmes, en particulier est notoirement ardue. Dès 1982 Lamport, Shostak et Pease soulignent dans un célèbre article [58] qu'ils ne connaissent « aucun autre domaine de l'informatique ou des mathématiques dans lequel un raisonnement informel est plus susceptible d'être erroné ». Dans la pratique, la diversité de variantes, le manque de réutilisabilité des preuves même entre nuances très proches des hypothèses de travail handicapent le processus de validation des protocoles typiquement écrits dans un langage informel. Malgré tout le soin apporté par leurs auteurs on trouve encore dans la littérature des algorithmes ne respectant pas les propriétés qu'ils sont censés vérifier [1, 39].

Sans s'appuyer sur une certification formelle, on risque l'introduction d'erreurs subtiles dont les conséquences peuvent s'avérer catastrophiques une fois le système déployé. Ce risque est d'autant plus prégnant que le domaine est émergent, à l'instar de la robotique en essaim.

C'est un des grands objectifs des méthodes formelles que de pouvoir garantir la sûreté de fonctionnement dans ce type de situation. Sans surprise, on trouve donc des tentatives d'application aux essaims de robots.

Le Model-Checking, également à l'œuvre dans la synthèse automatique de protocoles, a par exemple été utilisé avec succès. Très automatisé et présentant l'avantage de produire systématiquement un contre-modèle dans les cas incorrects, il a permis de découvrir des erreurs dans différents travaux publiés, dans un contexte toutefois relativement simple où les robots évoluent dans un espace discret. Il s'applique cependant sur des *instances* de problèmes et ne permet pas de raisonner en toute généralité. Typiquement, montrer qu'il n'existe aucun protocole capable de remplir une tâche donnée sous certaines hypothèses n'est pas à sa portée.

Une autre méthode, la preuve formelle, peut être vue comme son complémentaire. Cette approche fait l'usage d'assistants à la preuve : des outils logiciels qui ne trouvent pas seuls une preuve en complète automatisation mais vérifient qu'une preuve qu'on leur fournit est bien correcte. Coq est l'un de ces assistants ; muni d'une logique d'ordre supérieur, il bénéficie d'un grand pouvoir d'expression et permet de manipuler dans un même contexte données, calculs, propriétés et preuves de ces propriétés. L'automatisation est faible, on sera en revanche capable d'énoncer, établir et finalement certifier corrects des résultats très généraux, comme ceux d'impossibilité cités plus haut, y compris dans des espaces continus.

Les travaux de cette thèse abordent la preuve formelle de propriétés d'essaims de robots. Ils visent à permettre la vérification, de protocoles ou de résultats théoriques plus abstraits, sous des hypothèses de travail présentant un caractère plus réaliste que l'existant.

Le parti pris est délibérément informatique. Nous nous intéressons en particulier à la description d'un modèle pour les essaims de robots proposée par Suzuki & Yamashita [74]. Le modèle original décrit des robots se déplaçant dans un espace en fonction de leur perception de l'environnement, selon un cycle de trois phases :

1. *Look* durant laquelle les robots observent un instantané de leur environnement ;
2. *Compute* durant laquelle la destination est calculée à l'aide du programme embarqué, le protocole ;
3. *Move* durant laquelle le robot se déplace vers la destination calculée.

Le modèle de Suzuki & Yamashita est remarquable par la diversités de ses variantes ; les principales questions qui s'y rapportent concernent la détermination de bornes : quelles sont les hypothèses les plus faibles sous lesquelles une tâche reste réalisable ?

Très succinct, épuré, dans sa définition initiale, il laisse en effet une grande liberté dans le choix des hypothèses : sur l'espace bien sûr et sa topologie, sur la nature et les capacités des robots, en particulier de leurs capteurs et de leurs actuateurs, enfin sur l'environnement : en quelque sorte l'adversaire.

Les variantes rencontrées sont autant de réponses à un besoin de réalisme. La possibilité d'une réinitialisation « sauvage » dans un milieu fortement radioactif pourra justifier par exemple qu'on ne souhaite pas, dans certains cas, assujettir le succès d'une tâche à une mémorisation des états précédents. Il n'y a à l'origine pas d'hypothèse particulière sur la façon dont l'univers est perçu : les performances des capteurs à modéliser pourront motiver différents modes ou des portées de visibilité plus ou moins importantes. On voudra éventuellement prendre en compte des vitesses de déplacement, etc.

À l'étude des possibilités offertes par différentes hypothèses, nous ajouterons quant à nous la préoccupation : ce résultat est-il sûr ?

Notre approche de vérification s'appuie sur le cadre formel PACTOLE[3] développé pour l'assistant à la preuve COQ. PACTOLE est constitué d'une vaste bibliothèque formelle pour l'expression et la preuve de protocoles et de propriétés sur le modèle de Suzuki & Yamashita. Initié en 2009, ce cadre formel a montré la faisabilité d'une approche par preuve dans le contexte des robots mobiles. Il a permis de certifier non seulement des résultats de correction de protocole [32, 11] mais aussi d'*impossibilité* [31] y compris en présence de fautes byzantines (c'est-à-dire comportement arbitraires voire hostiles de certains robots défaillants) [6]. Le prototype au commencement de nos travaux considérait toutefois des hypothèses assez simples : espace Euclidien, robots ponctuels, sans volume, ordonnancement totalement ou semi-synchrone, etc.

Nos contributions étendent ce cadre formel. Par des facilités de spécification, par un enrichissement du cœur du modèle à une plus grande variété de domaines, elles introduisent des hypothèses toujours plus réalistes et s'inscrivent donc dans une volonté de rendre accessible au plus grand nombre de développeurs des techniques de vérification formelle fondées sur la preuve. Nous proposons en particulier un cadre d'expression des espaces discrets, c'est-à-dire ici de *graphes*, domaine très étudié en pratique. Nous ajoutons aux synchronisations en place la possibilité de modéliser des exécutions totalement *asynchrones* et donc bien plus réalistes. Nous permettons enfin de travailler sur des robots qui ne sont plus simplement des points sans épaisseur mais qui présentent un *volume* et des risques de collisions.

Au delà de l'exercice technique, nous illustrons systématiquement l'utilisabilité pratique de nos formalisations à travers différentes mises en œuvre : preuve d'impossibilité d'exploration dans les anneaux, preuve d'équivalence entre deux représentations des déplacements dans des graphes, conservation d'invariant pour un protocole distribué de poursuite et maintien de connexion avec une cible mobile.

Ce document présente nos travaux, il est organisé comme suit : le chapitre 2 détaille le modèle proposé par Suzuki & Yamashita [74]. Nous y détaillons le cœur du modèle, les particularités des robots, ainsi que différents degrés de liberté définissant des variantes. Nous exposons également certains des principaux problèmes étudiés dans ce contexte.

Pour maîtriser les erreurs malheureusement encore rencontrées dans des travaux pourtant très soigneux, nous nous intéressons aux méthodes formelles dont nous donnons un aperçu au chapitre 3. Le spectre des méthodes formelles est large ; on rencontre principalement en algorithmique distribuées des approches de spécifications outillées par des assistants mécaniques, du model-checking, et depuis peu de la preuve formelle sur laquelle se concentrent nos travaux, au travers de l'approche PACTOLE.

Le chapitre 4 donne un bref exposé de l'assistant à la preuve COQ et décrit le cœur de PACTOLE. Il présente comment le modèle de Suzuki & Yamashita est fidèlement reproduit dans le cadre formel. PACTOLE a connu plusieurs phases de développement, nous faisons dans ce document, par souci de cohérence, le choix de décrire sa version la plus récente utilisant les « typeclasses » de COQ et dans laquelle ont été portées nos contributions initialement développées pour une architecture de modules.

Le chapitre 5 présente la première de nos contributions avec l'extension de PACTOLE aux graphes, nous illustrons une première fois la validité de cette extension avec une preuve d'impossibilité en certifiant

[3] https://pactole.liris.cnrs.fr

formellement qu'on ne peut réaliser l'exploration avec stop d'un anneau lorsque le nombre de robots (sous certaines hypothèses de capacités) divise le nombre de nœuds de l'anneau. Un article présentant ces travaux a été publié dans les actes de la conférence internationale ICDCN2018 [12].

Le chapitre 6 quant à lui explique notre extension du noyau de PACTOLE au cas de séquencement totalement asynchrone. Nous montrons la pertinence de notre modélisation en la mettant en œuvre dans une preuve d'équivalence entre deux modèles de nouveau sur les graphes : un modèle purement discret où les agents sautent de nœud en nœud et un modèle plus réaliste où les robots se déplacent continuement le long des arêtes mais ne sont perçus que comme étant dans un voisinage des extrémités de celles-ci. Ces travaux ont été présentés et publiés dans les colloques internationaux SSS2018[7], NETYS2019 [8] et ALGOTEL2020 [9].

Le chapitre 7 aborde le traitement des robots qui ne sont plus simplement des points dans un espace mais qui présentent des risques de collision. Dans une étude de cas nous développons un algorithme de poursuite de cible inspiré de travaux de Reynaud [70] : il s'agit d'assurer qu'une communication est toujours établie entre une base et une équipe de recherche (cible) à l'aide d'un réseau de drones servant de relais. Nous proposons un protocole totalement synchrone pour cette tâche, décrivons les étapes de sa spécification dans notre cadre formel et garantissons formellement l'existence à chaque instant d'un lien de communication entre la base et la cible. Nous dotons finalement nos robots d'un volume dans une extension simple de cette étude.

Le chapitre 8 conclut notre exposé et propose quelques pistes d'exploration futures.

Chapter 2

Un modèle pour l'étude des essaims de robots

Nous présentons le modèle introduit par Suzuki & Yamashita [74]. Simple dans sa description, conçu pour une analyse des essaims des points de vue informatique et algorithmique, ce modèle s'est enrichi de nombreuses variantes. On peut en résumer le principe fondamental en énonçant que *des robots se déplacent dans un espace vers une destination calculée en fonction de leur perception de l'environnement*. Nous allons donc décrire dans un premier temps les espaces considérés, les caractéristiques et capacités usuellement rencontrées pour les robots et en particulier celles de leur mode de perception. Nous présenterons enfin les différents aléas et modes de synchronisation qui peuvent décrire les évolutions du système.

Nous détaillerons ensuite quelques-uns des principaux problèmes, fondamentaux ou plus appliqués, étudiés dans ce cadre.

2.1 L'espace

On distingue principalement deux grandes familles d'espaces dans la littérature sur les robots mobiles, les espaces continus d'une part, et les espaces discrets d'autre part.

Les espaces continus. Dans la plupart des travaux sur les espaces continus, ces derniers sont euclidiens[1] ; les positions des robots sont simplement des points ou des voisinages quelconques. La trajectoire des robots peut être spécifiée, ou définie par le programme embarqué dans chaque agent. Sauf mention contraire, on considère cependant des trajets rectilignes.

Nous travaillerons plus particulièrement avec ces espaces au long du chapitre 7.

Les espaces discrets. Les positions privilégiées (destinations) possibles des robots dans ces espaces sont discrètes et distinguées, leurs liaisons spécifiées. On est essentiellement dans le cas de graphes (où les positions privilégiées sont les nœuds). Le chemin entre deux positions dans le graphe est classiquement dans ce cas la liste des sommets séparant les deux positions.

Un cas hybride. On peut toutefois, par souci de réalisme, considérer certains espaces exhibant des caractéristiques de ces deux grandes familles, par exemple en considérant des arêtes de graphes continues de dimension 1 dont les points sont des positions possibles (c'est-à-dire traversables au cours

[1] La littérature étudie essentiellement ces espaces en une ou deux dimensions.

Rapport-Gratuit.com

d'un déplacement) mais non privilégiées des agents qui visent les nœuds. La description d'un chemin comprend alors également la liste des arêtes empruntées.

Nous nous intéresserons aux graphes discrets et hybrides au cours des chapitres 5 et 6.

2.2 Les robots

La diversité des situations auxquelles les robots peuvent être confrontés dans des applications pratiques, en particulier du fait de l'adversité supposée de l'environnement, invite à la prudence quant aux hypothèses sur lesquelles se reposer : peut-on faire confiance à d'éventuelles communications ? les agents vont-ils se réinitialiser du fait de radiations, par exemple et peut-on alors faire confiance à leur mémoire ? etc. Les capacités et caractéristiques des robots à modéliser varient donc en fonction de la résilience désirée.

Dans sa version fondamentale et initiale, le modèle de Suzuki & Yamashita pose cependant un cadre commun.

Les robots sont homogènes. Ils présentent les mêmes caractéristiques internes et externes au sens où ils sont construits sur le même modèle.

Les robots sont autonomes. Ils constituent un système distribué décentralisé : chacun des robots prend lui-même ses décisions, elles ne lui sont pas dictées par une quelconque unité centrale.

Les robots suivent le même algorithme. La prise de décision dépend d'un algorithme commun à tous les robots : le *protocole*. Ce protocole peut être suivant les modèles *déterministe* ou *non déterministe*.

Il n'y a pas de canal de communication explicite. Les robots sont dit *silencieux*. Les seuls échanges d'informations se font via l'observation de l'état perceptible des autres robots, par exemple leurs positions relatives et éventuellement d'autres modalités visibles comme l'affichage de signaux colorés.

Ce silence des robots peut être motivé par les applications ou les environnements hostiles dans lesquels les perturbations de communications sont possibles.

La perception de l'environnement est un instantané. Les états perceptibles reçus sont ceux à un instant donné.

La prise de décision, le résultat de l'algorithme *ne dépend que de la perception* des robots. En particulier si deux robots ont des perceptions équivalentes (pour une certaine définition) les ensembles de réponses sont équivalents. Par exemple deux robots au protocole déterministe, situés à la même position et recevant la même perception de l'environnement, calculeront le même état résultat (comprenant la même destination).

Les principales variantes du modèle portent sur les caractéristiques des robots et de leurs capteurs.

Anonymat. Les robots sont tous construits sur le même modèle mais cela n'empêche pas une forme d'identification. Le modèle original [74] considère des robots *indistinguables* au sens où leur propre perception ne sait les différencier et *anonymes*. C'est le cadre où nous nous placerons dans la majeure partie de nos travaux. Nous modéliserons et utiliserons cependant des robots avec identifiant lors de notre étude de cas complète du chapitre 7.

Persistance de la mémoire. On peut considérer que les robots ont une certaine quantité de mémoire pour l'exécution du protocole. Cela ne signifie pas qu'ils sont capables de retenir des informations (perceptions, états) de leur actions passées.

On parle de robots *oublieux* lorsque leurs prises de décision et changements d'état ne peuvent pas dépendre des actions passées (en plus de la perception actuelle). C'est le cas des robots du modèle initial ; la contrainte est lourde mais légitimée par les cas pratiques où des réinitialisations sont à craindre.

Certains travaux supposent toutefois des quantités de mémoire persistante et étudient la réalisabilité de certaines tâches en fonction de ces quantités [15, 14, 74].

Volume, collisions. Les hypothèses les plus faibles considèrent les robots comme

- *Ponctuels* (ils n'ont pas d'épaisseur),
- *Transparents* (ils ne masquent pas la perception),
- *Sans collision* (deux robots peuvent se trouver au même endroit exact, se croiser sans problème).

Le souci de réalisme a bien sûr amené à revisiter ces propriétés et proposer des *robots volumiques*. Deux robots volumiques peuvent se toucher mais pas se superposer. Si jamais lors d'un mouvement deux robots venaient à rentrer en collision, ils seraient considérés comme devenant fautifs. La perception de ce volume entraîne le partage d'une unité de distance commune ; ces robots peuvent être vus comme des boules d'un diamètre donné.

Notons que la notion de volume n'entraîne pas *forcément* l'opacité des robots et il existe des études sur des robots volumiques mais transparents [29, 34, 47].

Nous proposons une modélisation formelle et une étude de cas comportant des robots volumiques dans notre chapitre 7.

Orientation. Les robots partagent certes le même espace mais ils n'en partagent pas forcément l'orientation. Dans le cas des espaces de type euclidien, on distinguera :

- Aucun partage de référentiel : chaque robot a alors son propre référentiel **dans lequel il se situe lui-même à l'origine** 2.1a.
- Partage d'axe : les robots sont d'accord sur une direction donnée, en faire un axe à part entière dépend alors d'un partage de *chiralité*, c'est-à-dire d'une notion de droite et gauche partagée 2.1b.
- Partage de référentiel : dans ce cas la description de l'espace est partagée par tous les robots 2.1c.

Dans le cas d'espaces discrets (les graphes), la notion de référentiel d'un robot peut être plus ardue à définir : les graphes ne comprenant pas forcément de système de coordonnées bien défini.

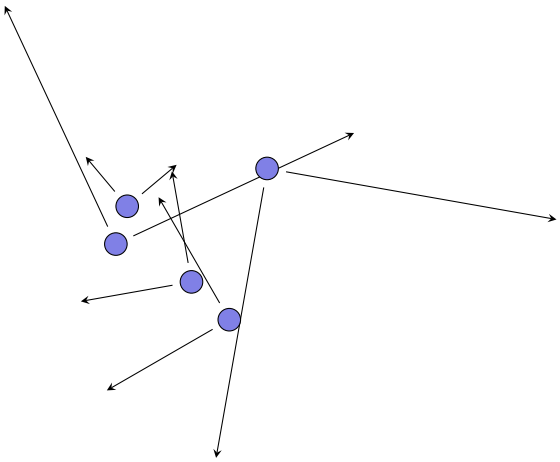
2.2.1 Capteurs, perception

Les capacités des capteurs, qui définissent la perception de l'environnement, offrent un large champ de possibilités pour des variantes du modèle de 1999.

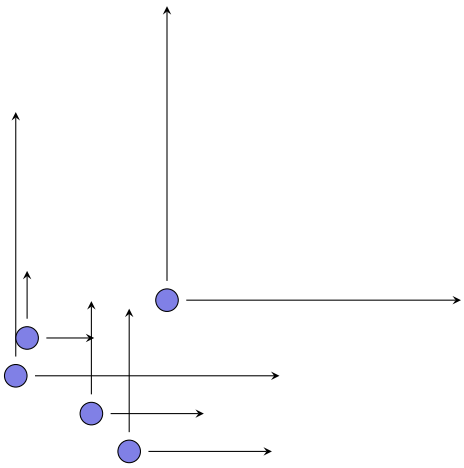
La *portée* des capteurs est un premier levier. On a souvent privilégié dans la littérature une vision globale de l'espace :

Définition 2.2.1 (Visibilité illimitée) *Le modèle à visibilité illimité est un modèle où l'instantané de la situation (*snapshot*) contient tous les états perceptibles.*

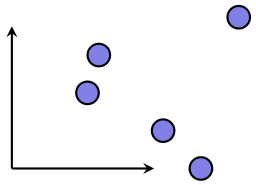
Son pendant est l'existence d'une sphère au-delà de laquelle un robot ne perçoit rien :



(a) Un exemple où les robots ne partagent aucune information sur un référentiel.



(b) Un exemple où les robots partagent deux axes.



(c) Un exemple où les robots ont un référentiel commun

Figure 2.1: Exemples de différentes possibilités de partage d'information pour la représentation des espace euclidiens.

Définition 2.2.2 (Visibilité limitée) *Le modèle à visibilité limitée est un modèle où l'instantané ne contient que les états perceptibles des robots situés jusqu'à une certaine distance du robot considéré.*

On appelle cette distance limite le *rayon de visibilité*. Ces modèles peuvent en outre être nuancés par l'éventuelle transparence des robots.

La notion de *graphe de visibilité* (au temps t) capture les relations de visibilité entre robots :

Définition 2.2.3 (Graphe de visibilité) *Le graphe $G(t) = (N, E(t))$ des robots au temps t est le graphe contenant l'ensemble N des robots et tel que $\forall\ r, s \in N, (r, s) \in E(t)$ si et seulement si r et s sont distants de moins de V, la distance de visibilité.*

La qualité de la captation est aussi à prendre en compte : un robot sait-il juste qu'une position est occupée ou peut-il déterminer le nombre exact de ses occupants ?

Définition 2.2.4 (Détection de multiplicité) *On appelle* détection de la multiplicité *la capacité des robots à compter le nombre de robot dans un même endroit.*

Dans les cas où cette détection est possible, les principales formes de multiplicité utilisées sont :

Multiplicité faible : on parle de détection de multiplicité faible quand le robot sait juste s'il y a plus d'un robot à une position donnée, sans en connaître le nombre exact.

Multiplicité forte : on parle au contraire de multiplicité forte, s'il est possible de savoir exactement le nombre de robots présents à un même point.

Détection locale : on parle de détection locale de la multiplicité quand le robot ne peut détecter la multiplicité qu'à sa propre position.

Détection globale : on parle enfin de détection globale de la multiplicité si le robot peut la détecter sur tout son champs de vision.

Ces différentes notions de détection de la multiplicité peuvent bien sur être combinées entre elles ; les différentes combinaisons ne sont cependant pas toujours comparables les unes aux autres. Il existe des problèmes résolubles en détection globale mais faible, qui ne le sont pas en détection locale mais forte, et inversement [23, 54, 67].

Les figures 2.2 et 2.3 illustrent les différences entre ces notions, respectivement dans un espace euclidien à deux dimensions et dans un graphe.

Volume et opacité. Une difficulté supplémentaire survient dans le cas de robots volumiques *opaques* où il peut devenir possible à un robot d'en cacher d'autres (voir figure 2.9). L'alternance des visibilités de certains robots complique alors l'utilisation d'invariants sur les positions perçues.

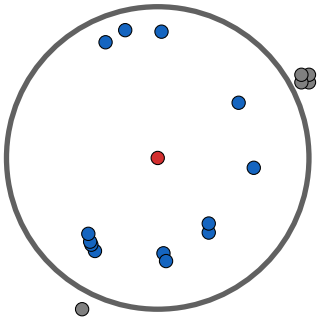
Le cas des robots lumineux. Tout en gardant l'idée de n'avoir aucun canal de communication explicite, on peut ajouter un affichage, disons une diode changeant de couleur, sur un robot.

Cette variante, introduite par Das et al. [35], ajoute à l'état perceptible de chaque robot une couleur qui varie en fonction de l'exécution.

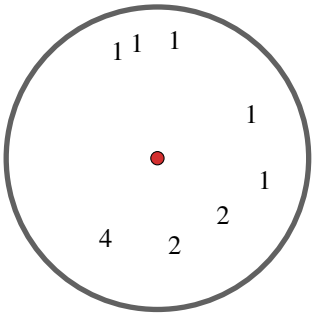
Là encore de nombreuses nuances sont possibles.

Par exemple cette lumière colorée peut être persistante ou non, c'est-à-dire ne pas être systématiquement réinitialisée à chaque activation.

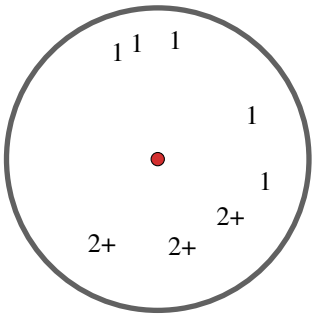
Une forme de mémorisation peut le cas échéant apparaître si le robot peut percevoir sa propre couleur persistante, changeant alors dramatiquement la puissance du modèle. Nous évoquons cet aspect à la fin de ce chapitre, une fois définis les modes de synchronisation et les problèmes étudiés.



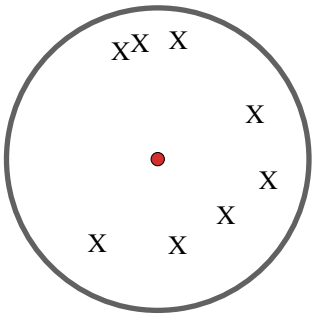
(a) Une configuration.



(b) Multiplicité globale forte.

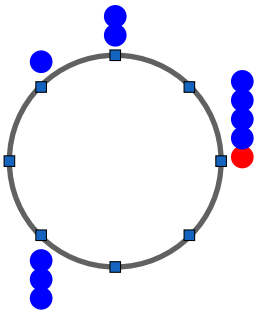


(c) Multiplicité globale faible.

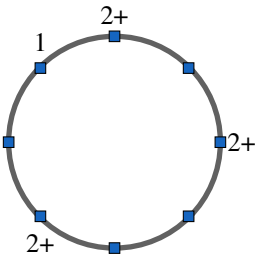


(d) Sans multiplicité.

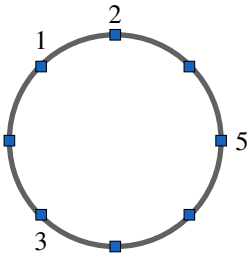
Figure 2.2: En (a) est représenté par un cercle l'espace visible par le robot central ●, les robots ● sont en dehors de son champ de vision mais les robots ● sont bien détectés. En (b) sa perception est donnée avec une détection de multiplicité globale forte, où le nombre exact de robots à un point est connu. En (c) se trouve cette même perception mais cette fois avec une détection de la multiplicité globale faible : le robot ● sait juste s'il y a un unique robot ou s'il y en a plusieurs. Enfin en (d) notre robot central ne bénéficie pas de détection de multiplicité, il sait juste qu'il y a au moins un robot à certains endroits.



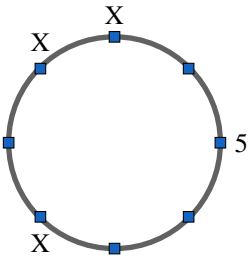
(a) Une nouvelle configuration de robots dans un graphe.



(b) Multiplicité globale faible.



(c) Multiplicité globale forte.



(d) Multiplicité locale forte.

Figure 2.3: La configuration en (a) est un graphe avec ● le robot observant et ● les autres robots. En (b) et en (c) nous avons des exemples de détection de la multiplicité globale dans ce graphe. Enfin en (d) nous avons un exemple de multiplicité locale.

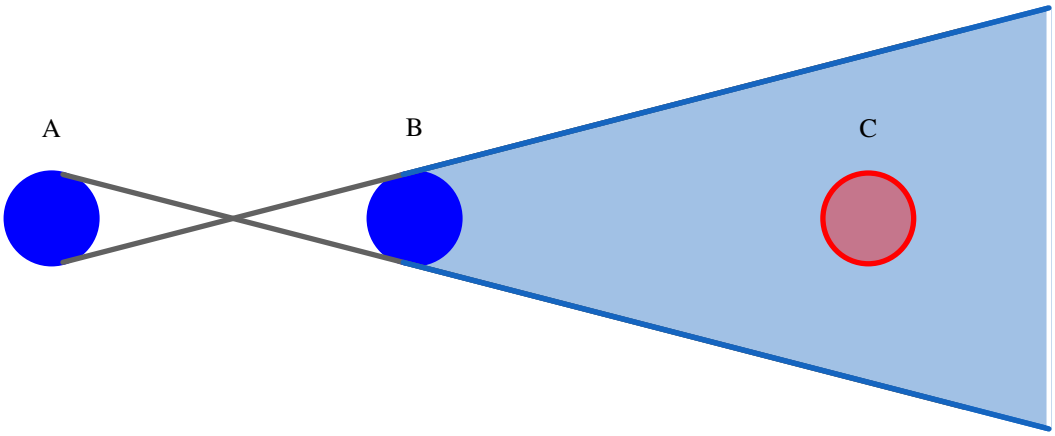


Figure 2.4: Le robot A ne voit pas le robot C car le robot B bloque la vision dans la zone à fond bleu ▭.

2.2.2 Le comportement

Au cœur de leur modèle, Suzuki & Yamashita décrivent la façon dont les robots agissent et interagissent comme suit : 1) un robot est soit actif soit inactif, 2) quand il est actif, le robot effectue un cycle de trois phases:

1. *Look*  Le robot fait un snapshot de l'espace observé recevant la position des autres robots (plus précisément leur état perceptible) relativement à sa propre position.

2. *Compute*  En utilisant le snapshot fait dans la phase *Look*, le robot calcule grâce à son programme la position de sa prochaine destination (plus précisément son prochain état).

3. *Move*  Le robot est en déplacement jusqu'à la position calculée. Si cette position est sa position actuelle, il reste sur place.

Comme précisé au paragraphe précédent, la portée et la précision du snapshot peuvent varier en fonction des hypothèses. Dans les cas courants où les robots sont vus comme des points, leurs positions dans le snapshot peuvent par exemple être leurs coordonnées dans le référentiel du robot r qui les observe et dont l'origine est la position de r.

La fin du cycle est déterminée par le passage à l'état inactif ou à la réactivation du robot, *elle ne signifie pas que les destinations calculées sont atteintes*.

Ce cycle de *Look*, *Compute* et *Move* (abrégé en cycle LCM) est répété indéfiniment.

Remarque 1 *Le fait que le snapshot dans la phase* Look *soit instantané ne pose pas de problème lors du passage à des variantes plus générales : le temps passé à la capture effective du snapshot (c'est-à-dire activation des instrument et des capteurs) ou au traitement des informations reçues desdits capteurs peut être considéré respectivement comme un temps d'inactivité ou comme faisant partie du temps de calcul dans* Compute*.*

L'ensemble des informations sur les états et emplacements des robots dans l'espace constitue une *configuration*.

Définition 2.2.5 (Configuration) *Une configuration est la description dans l'espace de la position des robots et tous leurs caractéristiques et paramètres.*

Elle permet en particulier d'avoir accès à ces information en fonction d'un identifiant de robot.

2.3 Caractéristiques de synchronisation

La notion de temps dans l'algorithmique sur les robots mobiles est liée à l'évolution du système : dans chaque modèle, la division du temps se fait en fonction des changements pertinents dans la configuration. Par *pertinents* nous voulons nous limiter aux changements incluant une modification de phase d'au moins un robot.

La séquence (infinie) des configurations liées à ces changements pertinents définit une *exécution*.

Définition 2.3.1 (Exécution) *Une exécution est la suite des configurations pertinentes du système dans le temps.*

Elle caractérise l'évolution du système.

Les exécutions sont habituellement rangées en trois familles principales, lesquelles peuvent être de nouveau subdivisées en fonction de contraintes additionnelles : les exécutions *totalement synchrones* (FSYNC pour Fully Synchronous), *semi synchrones* (SSYNC pour Semi Synchronous) et enfin *asynchrones* (ASYNC).

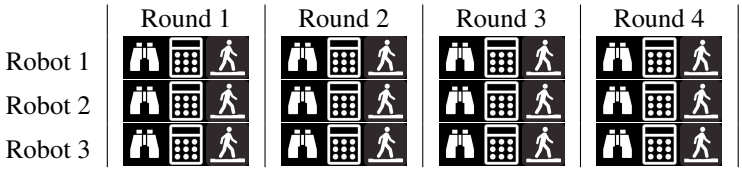


Figure 2.5: Une exécution FSYNC: on y voit les trois phases formant un round ; tous les robots commencent ce round en même temps.

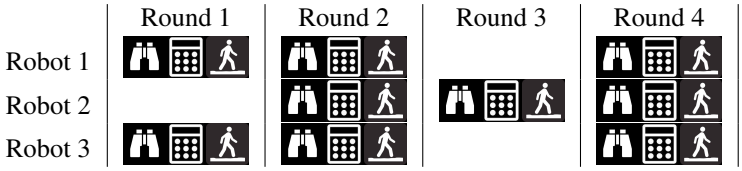


Figure 2.6: Le comportement SSYNC: un sous ensemble des robots sont activés à chaque round. Le robot 2 n'est en particulier pas activé lors du premier round ; les robots 1 et 3 ne sont pas activés lors du troisième round.

Remarque 2 *En reprenant la notion de démon de Maxwell, on parlera de* démon *caractérisant une exécution.* C'est l'adversaire qu'on suppose ayant choisi les activations, les fautes, les changements de référentiels, etc. *Il ne s'agit en rien d'une centralisation du système. Nous utiliserons dans la suite cette notion de démon comme rassemblant les caractéristiques des exécutions que ses choix entraînent.*

Les cas FSYNC et SSYNC ont en commun l'atomicité du cycle LCM où chacune des phases commence en même temps pour tous les robots.

Définition 2.3.2 (Exécution FSYNC) *Une exécution totalement synchrone est une exécution où le cycle* LCM *est atomique et dans lequel tous les robots sont activés et exécutent chacune de leurs phases en même temps.*

Essentiellement : on peut diviser l'exécution en *rounds* et chaque début de round correspond à l'activation de tous les robots.

La figure 2.5 illustre cette famille d'exécutions.

Un relâchement sur la contrainte que *tout* robot participe à *chaque* round permet de définir le cadre SSYNC.

Définition 2.3.3 (Exécution SSYNC) *Une exécution semi-synchrone est une exécution où le cycle* LCM *est atomique et dans laquelle un sous-ensemble des robots sont activés et exécutent chacune de leurs phases en même temps.*

Remarque 3 *Si ce sous-ensemble comprend systématiquement tous les robots alors on se retrouve dans le cas d'une exécution* FSYNC.

La famille d'exécutions la plus réaliste ne requiert plus l'atomicité : on parle alors d'exécution *asynchrone* (ASYNC).

Définition 2.3.4 (Exécution ASYNC) *Une exécution* ASYNC *est une exécution où le cycle* LCM *n'est plus atomique et peut être arbitrairement découpé. Les activations et changements de phases d'un robot sont indépendants des états des autres robots.*

Le calcul d'un nouvel état, d'une destination, peut dans le cas ASYNC reposer sur une perception obsolète : les robots observés ayant pu bouger durant la phase de calcul.

Ce modèle est équivalent à un modèle où il existerait des périodes d'inactivité entre chacune des phases du cycle LCM, car ces temps d'inactivité peuvent être assimilés au temps de calcul ou à un début de déplacement où le robot ne bouge pas encore [46].

La figure 2.7 illustre une telle exécution avec une longue phase *Look* durant laquelle ont lieu des déplacements d'autres robots.

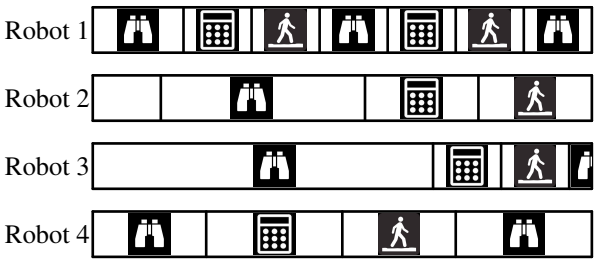


Figure 2.7: le comportement ASYNC: chaque phase peut recouvrir d'autres phases des autres robots, il est donc possible de prendre des décisions sur des informations périmées. Par exemple durant la phase *Compute* du robot 4, le robot 1 a pu bouger : les informations du robots 4 ne sont plus à jour. Similairement le snapshot du robot 3 a pu être pris avant ou après les déplacements du robots 1, il est possible que le calcul de 3 se fonde sur une situation qui n'est plus.

Remarque 4 *On peut déterminer une forme de hiérarchie dans les synchronisations et en particulier :*

- *Toute exécution* FSYNC *peut être vue comme une exécution* SSYNC *où le sous-ensemble activé à chaque round est l'ensemble complet des robots.*
- *Toute exécution* SSYNC *peut être vue comme une exécution* ASYNC *où les phases des différents robots se déroulent en même temps et atomiquement, respectant ainsi l'idée de round.*

Raffiner les familles : équité dans les exécutions.

Toutes les exécutions ne sont pas pertinentes lorsqu'on s'intéresse à la réalisation d'une tâche et on peut vouloir éviter les cas limites où par exemple un robot n'est jamais activé. On définit pour cela la notion d'*équité* (ou *fairness*).

Un démon équitable est un démon qui, dans une exécution infinie, activera n'importe quel robot une infinité de fois, mais sans certitude de temps (au sens de nombre d'activations des autres robots) entre deux de ses activations.

Définition 2.3.5 (Démon équitable) *Un démon est dit équitable si pour tout robot* r*, et tout instant* t *des exécutions qu'il caractérise, il existe un instant* t' *tel que* $t \leq t'$ *et auquel* r *est activé.*

Cette notion peut elle-même être contrainte par le nombre, entre deux activations de chaque robot, des activations de ses congénères [31].

Définition 2.3.6 (Démon k-équitable) *Un démon est dit k-équitable s'il est équitable et qu'entre deux activations de tout robot* r*, tout autre robot sera activé au plus k fois.*

Les différents démons de la littérature n'ont pas tous le même pouvoir au sens où certaines tâches sont réalisables avec certains et pas avec d'autres. Ils ne forment cependant pas un treillis complet [40].

Les fautes

Les grandes applications des essaims de robots prévoient d'envoyer ces derniers dans des zones dangereuses, potentiellement soumises à des rayonnement dangereux. Il est donc possible que leurs mémoires soient endommagées voire que ces robots ne suivent plus le protocole. Ces cas font partie des choix que l'ont peut autoriser le démon à faire.

Dans le cas où un robot s'arrête juste de fonctionner on parle de faute *crash* [80], et là où certains robots peuvent exhiber un comportement arbitraire, on parle de fautes *byzantines* [2].

On dira enfin d'un protocole qu'il est *tolérant aux fautes* (pour une certaine catégorie de fautes) s'il peut résoudre un problème malgré la présence de robots fautifs (en général quantifiés en proportion du nombre total de robots).

Le cas des déplacements

Des hypothèses peuvent enfin être posées sur la réalisation complète ou incomplète des déplacements prévus. C'est bien la réactivation qui détermine la fin d'un cycle et non pas l'arrivée à la destination prévue.

On distingue donc deux variantes suivant la possibilité de réactiver un robot en chemin.

Les déplacements rigides atteignent toujours leur but :

Définition 2.3.7 (Déplacements rigides) *Les déplacements des robots sont dits rigides si le démon ne peut les interrompre et les réactiver avant qu'ils aient atteint leur destination calculée.*

Les déplacements flexibles sont interruptibles. Pour éviter des contre-exemples triviaux, basés sur le paradoxe de Zénon, nous imposons une distance minimale δ en deçà de laquelle les robots ne peuvent pas être réactivés :

Définition 2.3.8 (Déplacements flexibles) *Les déplacements des robots sont dits flexibles (ou δ-flexibles) si le démon peut interrompre et réactiver un robot au cours de son déplacement après que ce dernier a parcouru une distance minimale non nulle fixée δ.*

2.4 Accomplir une tâche, principaux problèmes étudiés

La littérature dans la thématique des robots mobiles est principalement dirigée par les problèmes : on y fixe un problème et on étudie sous quelles hypothèses (sous quelles variantes du modèle) ce problème est, ou non, résoluble.

Un protocole résout un problème si dans *toutes* les configurations initiales acceptables, ce protocole résout le problème quels que soient les choix du démon[2]. L'important ici c'est qu'il ne peut pas y avoir de cas où le protocole ne résout pas le problème : si un protocole ne fonctionne que pour certaines configurations initiales par exemple, ou certains choix, il ne résout pas le problème.

Similairement, un problème n'est pas résoluble si *pour tout protocole* il existe une configuration de départ et une suite de décisions du démon faisant échouer le problème.

[2]Ces choix sont bien sûr contraints par les hypothèses.

On peut caractériser les conditions pour lesquelles une tâche est réalisable de deux façons : les résultats positifs identifient des conditions *suffisantes* à la réalisation de la tâche en fournissant des protocoles qui résolvent cette tâche suivant certaines conditions. Inversement, les résultats négatifs cherchent à identifier des conditions *nécessaires*, en démontrant qu'en deçà aucun protocole ne peut résoudre la tâche.

Lors des résultats négatifs, la hiérarchie précédemment établie sur les démons peut être vu comme inversée. Un contre-exemple utilisant un démon FSYNC sera plus puissant qu'un contre-exemple utilisant SSYNC, car le contre-exemple FSYNC est également SSYNC. De même, un contre-exemple SSYNC est plus puissant qu'un contre-exemple ASYNC.

Les problèmes principalement étudiés sont fondamentaux ou plus appliqués, ils se rangent en deux grandes familles : d'abord la création de figures particulières (*pattern formation*), incluant notamment le cas fondamental particulier du rassemblement (*gathering*) en un seul point, ensuite les problèmes d'exploration, où les robots parcourent tout l'espace disponible, que ça soit pour récolter des informations (cartographie, patrouilles) ou pour en distribuer.

2.4.1 Formation de figures

Le problème de la création de figures est le problème où le but est de recréer en temps fini dans l'espace grâce à la position des robots une figure donnée en paramètre au programme et y rester. La figure donnée peut être une série de points dans l'espace, ou une figure géométrique précise (un hexagone régulier par exemple), l'inscription dans un cercle, etc.

Une variation sur la création de figure est le déplacement en formation *flocking*, où une fois une figure créée, les robots se déplacent en conservant la forme de la figure. Ici encore, deux variantes existent, celle où les robots suivent une cible extérieure, très étudiée en robotique [61, 65, 77, 78], celle où, sans cible, les robots connaissent la trajectoire à prendre [24, 72, 79, 71].

Dans le cas de robots anonymes, deux robots à exactement la même position peuvent avoir la même perception, suivant les informations (référentiel, par exemple) données par le démon. Cela amène ces robots à prendre exactement les mêmes décisions : ils peuvent donc rester indistinguables l'un de l'autre[3]. Dès lors, il est généralement supposé que la configuration initiale des problèmes de rassemblement ne contient que des robots isolés, sauf bien sûr si le contraire est précisé.

Lorsque plusieurs robots sont autorisés à avoir la même position, on parle de création de figure avec multiplicité. Ce problème de création a été largement étudié notamment pour les possibilités engendrées par le positionnement en figure, qui permet d'attribuer des rôles (par exemple une élection de leader) et des actions particulières [4, 42, 73, 77].

Ce problème a été également résolu en ASYNC mais en incorporant une mémoire aux robots [21].

Un cas particulier fondamental : le rassemblement

La création de la figure à un seul point définit le *rassemblement* (*gathering*).

Définition 2.4.1 (Problème du rassemblement) *Le problème du rassemblement consiste, à partir de robots disposés aléatoirement dans l'espace, à amener en temps fini tous ces robots en un seul point de l'espace, non prédéfini, et les faire y rester.*

Dans le cas où les robots sont au nombre de deux, on parle de *rendez-vous*.

Ce problème est fondamental au sens où il sert de préliminaire à la réalisation d'autres tâches, en maîtrisant une configuration particulière et en caractérisant un point de l'espace.

Comme dit précédemment, plusieurs travaux ont déjà été effectués sur cette problématique. Dans le cas de robots anonymes, oublieux, dans un espace euclidien sans référentiel partagé, le problème du

[3] Dans le cas de protocoles non-déterministes, rien n'empêche le démon de dicter les même tirages aléatoires.

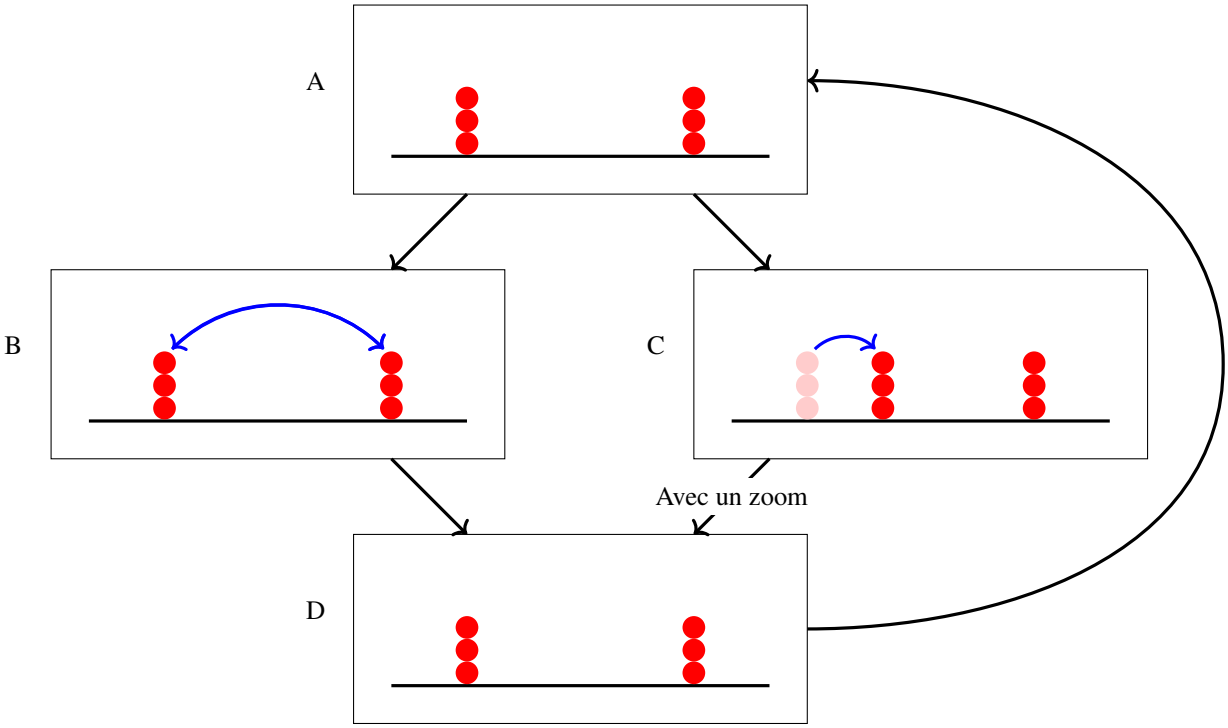


Figure 2.8: Exemple de configuration cyclique pour le problème du rassemblement. En partant d'une situation symétrique (A), il y a deux choix.

- Soit le protocole fait en sorte qu'une tour bouge à la place de l'autre, dans ce cas, le démon activera les deux tours au même moment. Comme la configuration est symétrique, le démon peut faire en sorte que les deux tours perçoivent la même chose, et donc les tours échangeront de place, et l'on revient au point de départ (D).
- Soit le protocole n'envoie pas une tour exactement à l'emplacement de l'autre, et dans ce cas là, le démon n'activera qu'une des deux tours (C), et en changeant d'échelle (ce qui est un choix du démon), on retrouve encore une fois la position de départ (D).

rassemblement de deux robots est impossible à résoudre en SSYNC (et donc en ASYNC) [74], voire celui d'un nombre pair de robots si on autorise une situation initiale avec deux points à multiplicité (tours) [31]. La figure 2.8 en expose l'argument de preuve.

De plus, même si il y a plus de 2 robots, si ces robots sont oublieux, anonymes et sans détection de multiplicité, quand aucune informations n'est partagées entre les robots, le problème est encore impossible [69].

Il est cependant possible de résoudre ce problème en FSYNC avec un algorithme très simple: chacun des deux robots bouge à la moitié de la distance les séparant !

Si tous les robots partagent un même sens de l'orientation (une même chiralité), alors ce problème est résoluble en SSYNC et ASYNC, avec par exemple le protocole donnant toujours comme destination la position du robot le plus « en haut et à droite » dans le référentiel partagé. Ce problème a même été étudié en présence de robots fautifs [19] ou encore avec un sens de l'orientation partagé, mais soumis à des erreur d'alignement par exemple [52].

Convergence. La variante *convergence* du problème du rassemblement relâche la contrainte sur l'empilement des robots au même endroit. Le problème devient alors de les rapprocher arbitrairement près les uns des autres.

Définition 2.4.2 (Convergence) *Le problème de convergence consiste à garantir l'existence d'un point de l'espace tel que pour tout ε non nul il existe un point de l'exécution à partir duquel tous les robots restent à une distance inférieure à ε de ce point.*

Cette variante est plus facilement résolue que le rassemblement. Pour les robots oublieux anonymes dans $\mathbb{R}^2$ elle admet par exemple une solution adaptable en ASYNC [26] au principe assez simple : il s'agit de se rapprocher du barycentre des robots.

Dispersement. Le problème inverse du rassemblement est le problème du dispersement (*scattering*), où le but est d'occuper l'espace avec une certaine répartition. Les problèmes les plus simples de cette classe reviennent à briser la multiplicité existante comme condition préliminaire à d'autre problèmes [38, 53], tandis que des problèmes plus avancés requièrent une couverture d'un espace borné [27, 45].

2.4.2 Exploration, le cas des graphes

Le principe de l'exploration dans un graphe est de visiter tous les noeuds du graphe dans lequel les robots évoluent. L'exploration a deux principales variantes: L'exploration avec stop et l'exploration perpétuelle.

Exploration avec stop

Dans le cas de l'exploration avec stop, la position des robots n'évolue plus une fois que l'espace (ici le graphe) a été totalement exploré.

Définition 2.4.3 (Exploration avec stop) *Le problème de l'exploration avec stop d'un graphe consiste à garantir que tous les nœuds du graphe ont été explorés et que les robots ne se déplacent plus une fois cette exploration terminée.*

Cette exploration a été étudiée dans le cadre de l'arbre [43] et de l'anneau [44], en ASYNC et avec une vision illimitée. Dans ces études, une difficulté apparaît dans le fait que les sommets et les arêtes étant anonymes, beaucoup de situations font ressortir des configurations symétriques. Dans chacun des travaux, des stratégies complexes sont utilisées pour communiquer aux robots dans quelle étape de l'algorithme ils sont. Un des principaux axes d'étude sur l'exploration avec stop est de savoir le nombre minimum de robots nécessaires pour la réaliser.

Dans le cas de l'arbre, en général au plus n robots sont nécessaires pour un arbre à n sommets [43]. Pour l'anneau, si le nombre k de robots divise la taille n de cet anneau (son nombre de sommets), alors l'exploration avec stop est impossible avec des algorithmes déterministes [44] . Nous donnons une certification formelle de ce résultat au chapitre 5.

Une solution à l'exploration avec stop pour k robots (avec $k \geq 17$) et n robots a toutefois été donnée [44], si k et n sont premiers entre eux.

Il a également été montré, toujours dans un modèle déterministe, qu'explorer un anneau de taille paire avec moins de 5 robots était impossible, et que 5 robots étaient suffisants pour tout anneau de taille n impaire (si 5 de ne divise pas n) [56].

Dans le cas de protocoles probabilistes, l'exploration avec stop est possible avec seulement 4 robots [37], avec un algorithme optimal pour tout $n \geq 8$. Un algorithme probabiliste optimal pour $n < 8$ a lui été donné par Devismes [36]. À noter que dans le cas probabiliste, il n'existe plus de problèmes de symétrie, car avec une probabilité $q < 1$, deux robots avec les mêmes informations prendront des décisions différentes. Il n'y a en particulier plus de contrainte sur k et n.

Exploration perpétuelle

Définition 2.4.4 (Exploration perpétuelle) *Le problème de l'exploration perpétuelle d'un graphe consiste à garantir qu'en tout point de l'exécution, tout nœud du graphe sera visité en temps fini.*

Pour ce type d'exploration, l'anneau a été étudié dans le modèle ASYNC avec une visibilité illimitée par Blin et al. [20] , et ces travaux établissent que pour un anneau de taille 10 ou plus, 3 robots sont nécessaires et suffisants pour l'exploration perpétuelle de manière déterministe, alors que pour une taille inférieur à 10, il n'existe pas de protocole résolvant ce problème avec 3 robots.

Baldoni et al. étudient l'exploration perpétuelle dans le cas d'une grille orientée en FSYNC, avec une vision limitée [13] notamment sous l'angle de la relation entre champ de vision et nombres de robots nécessaires et suffisants pour résoudre ce problème.

2.4.3 Puissances de quelques variantes

Robots lumineux

Les modèles de robots lumineux utilisent pour la plupart un nombre de couleurs fini et leur puissance peut varier selon ce nombre. Un modèle ASYNC avec des couleurs peut ainsi gagner en puissance. Avec seulement 5 couleurs, Das et al. montrent [35] que leur modèle ASYNC est au moins aussi puissant qu'un modèle SSYNC sans couleur, c'est-à-dire qu'il peut résoudre les même problèmes, et potentiellement d'autres. Ces mêmes travaux établissent en outre que si jamais ces robots colorés pouvaient se souvenir d'une seule configuration, ils seraient strictement plus puissant que des robots FSYNC oublieux et sans lumières.

Robots volumiques opaques

Les robots volumiques opaques ont en particulier été étudiés dans le contexte du rassemblement. Toutefois, étant volumiques, ils ne peuvent pas être exactement au même endroit. La définition du rassemblement est donc adaptée à ce cadre et le rassemblement pour des robots volumiques devient la formation d'une figure géométrique dont tous les robots doivent être conscients de la fin de formation.

L'opacité de ces robots ajoute une difficulté : il est impossible pour un robot de savoir si un autre est caché par un des robots visibles (comme illustré par la figure 2.9). Le démon peut en particulier faire bouger les robots de telle sorte à en cacher certains à d'autres.

Il n'existe pas à notre connaissance dans la littérature de solution pour le rassemblement de plus de 4 robots volumiques opaques. Czyzowicz, Gąsieniec et Pelc proposent un protocole pour le cas où $n = 3$, et un autre pour le cas où $n = 4$ [34].

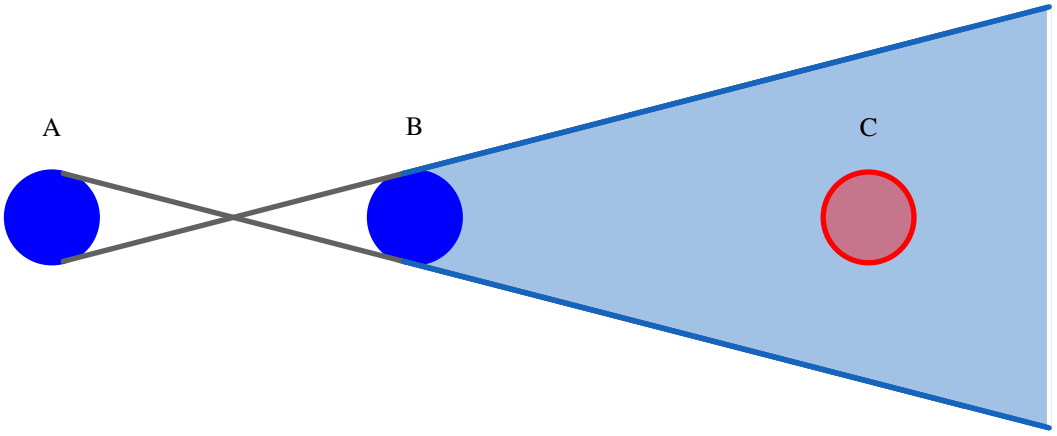


Figure 2.9: Le robot A ne voit pas le robot C car qu'importe où sont placés les capteurs de A, le robot B bloque la vision, dans la zone .

Chapter 3

Méthodes formelles

Le chapitre précédent a fait état de de travaux *démontrant* des résultats. Ces démonstrations sont cependant bien souvent faites « à la main », avec grand soin mais possiblement victimes d'erreurs humaines. De possibles variations subtiles entre le contexte et là où est justifié/appliqué un résultat invalident en effet ce dernier. On trouve malheureusement toujours des résultats incorrects dans la littérature [39] ; la plupart du temps ces erreurs sont découvertes grâce à l'apport des *méthodes formelles*.

Les méthodes formelles (il en existe plusieurs types) permettent d'énoncer formellement (sans aucune ambiguïté) les résultats et lemmes mais ont aussi pour but de permettre la vérification de la validité d'énoncés et de programmes. Cette vérification a lieu grâce d'une part, à des langages mathématiques non ambigus et d'autre part à des outils permettant cette vérification. Ces méthodes ont été utilisées avec succès dans différents domaines: mathématiques [49, 62], preuve de programmes [60], compilateurs certifiés[60], Système d'exploitation [55]. Les preuves formelles ont aussi eu un rôle dans l'industrie, avec la certification de la ligne du métro 14 à paris [59], ou encore la vérification de protocole d'atterrissage[64].

La grande variété dans les modèles de robots mobiles et la considérable difficulté à garantir la correction d'un algorithme distribué rendent leur utilisation nécessaire, en particulier dans les cas d'applications critiques.

L'algorithmique distribuée a bien sûr fait appel à ces méthodes, pour ne citer que quelques approches : au niveau des spécifications et raisonnement (possiblement outillés) avec la *Temporal Logic of Actions* de Lamport : TLA/TLA+Proofs [57, 33], au niveau de la vérification avec le model-checking et au niveau de l'algèbre des processus avec le π*-calcul* [63].

Le contexte des robots mobiles n'est pas parfaitement adapté à la modélisation TLA, plutôt pensée pour un modèle de processus avec partage de mémoire et envoi de messages. Les algèbres de processus ne sont pas non plus idéales pour notre modèle, car il n'y existe pas de notion de positions. Il a en revanche été fait largement usage du model-checking et de sa puissante automatisation, y compris pour la synthèse automatique de protocoles ; nous en donnons un aperçu au prochain paragraphe. Nous nous intéresserons pour notre part à une autre approche : la preuve formelle mécanisée. Pour plus de détail, il existe plusieurs études concernant les méthodes formelles appliquées aux robots mobiles[16, 68]

3.1 Model-checking

Parmi les différentes méthodes formelles, le *model checking* est une méthode très utilisée dans la vérification sur les robots mobiles grâce à son automatisation et sa relative facilité de prise en main.

Le principe de cette méthode est le suivant:

1. Représenter le système sous forme de système de transition. Cela amène donc une abstraction et

une sur-approximation du modèle de base. Cette sur-approximation contient cependant tout les états du système de base, ainsi les propriétés de sûreté sur le modèle de transition restent valables sur le système original.

2. Analyser tous les cas possibles lors de l'évolution du système de transition.

Ensuite il faut définir la propriété qui doit être vérifiée grâce à, par exemple, de la logique temporelle: la propriété *S* en langage naturel *pour toute la durée de l'exécution, il n'existe pas deux robot à la même position* peut se traduire en logique temporelle en $GF(\forall xy, x \neq y => pos(x) \neq pos(y))$. Cette propriété ϕ est une *abstraction* de *S* qui est créée sur mesure pour être vérifié de manière exhaustive et automatique par le *model-checker*.

ϕ doit ensuite rester valable. Si cette validité de ϕ est bien là, la propriété *S* est vraie car ϕ est une propriété de sûreté. De plus, si ϕ n'est pas vérifiée, alors le *model-checker* retournera un contre-modèle. Ce contre-modèle n'est pas à proprement parler un contre-exemple de *S*, car il se peut que l'abstraction de ϕ soit trop forte, et donc que le résultat ne corresponde pas exactement au problème *S*.

Cette méthode de vérification a pour avantage de n'avoir qu'à exprimer le modèle et le problème en système de transition d'états, et le logiciel donne une réponse automatiquement. Cela permet à des personnes non spécialisées dans le domaine des méthodes formelles de faire de la vérification formelle sur les modèles.

Cette méthode de vérification formelle a permis de prouver de nombreux résultats dans le cadre des robots mobiles utilisant des *instances* de problème. Pour citer quelques exemples : Bérard et al. proposent des preuves concernant la correction d'algorithmes pour résoudre l'exploration avec stop [17], ainsi que l'exhibition de contre-exemple pour le modèle perpétuel ; Défago donne et prouve correct un protocole pour le rassemblement entre deux robots lumineux [41].

Les outils utilisés sont extrêmement efficaces pour résoudre certains types de problèmes, notamment sur des instances particulières. Il est cependant plus difficile d'aborder par cette approche des problèmes généraux. En effet, comme il est nécessaire de modéliser toutes les possibilités, il existe une tension entre la précision de l'abstraction du problème et la taille à maintenir raisonnable des modèles à vérifier. Cette méthode est très susceptible à l'explosion combinatoire et donc peut devenir impraticable. Il est de même difficile de faire des vérifications sur l'impossibilité de résoudre un problème pour tout programme [1]. En résumé, l'utilisation de quantification universelle sur des ensembles importants ou infinis est souvent incompatible avec l'utilisation du model-checking.

3.2 Preuve formelle

La preuve formelle est l'approche suivie dans nos travaux. Elle consiste à prouver mathématiquement le problème de manière à ne laisser aucune place aux raisonnements implicites en se ramenant systématiquement aux règles de la logique établie pour cette preuve. Ce genre de tache étant extrêmement ardu, nous utilisons un outil, l'*assistant à la preuve*. Grâce au langage mathématique mais aussi au système interactif apportés tous les deux par ce genre d'outil, il est possible de donner une représentation purement logique et mathématique à un problème, pour ensuite réduire la vérification aux problèmes de correction et de complétude. Contrairement au model-checking, l'expressivité est très grande, et ainsi il est possible d'être plus précis dans l'écriture des énoncés. Cette précision apporte une plus grande confiance dans les énoncés car dans les assistant à la preuve, la syntaxe dans laquelle sont écrits les énoncés est très proche du langage mathématique, langage dans lequel sont écrits les énoncé à la base.

Il faut donc définir le modèle dans l'assistant à la preuve à l'aide de la logique et du langage de ce dernier. Une fois les modèles et les problèmes posés, il existe différentes approches suivant le type d'assistant à la preuve. Certains assistant ont un langage de preuve dédié, et la preuve y est décrite et

[1] C'est-à-dire qu'un problème ne peut être résolu qu'importe l'algorithme utilisé, voir paragraphe 2.4.

vérifiée. D'autre emploient un système de tactiques qui permettent de construire pas à pas un terme de preuve, qui devra vérifier l'énoncé.

- Ces tactiques changent dans un premier temps la preuve de l'énoncé en preuves d'énoncés plus simples.
- En enchaînant ces tactiques, ces assistant cherchent à se réduire aux hypothèses de l'énoncé de départ.
- En découle un arbre de preuve témoignant des règles de la logique utilisée par l'assistant pour construire la preuve.

Parmi les assistants à la preuve les plus utilisés nous retrouvons HOL light [50], Isabelle [66], Mizar [76], F* [75] et l'assistant Coq utilisé dans nos travaux et présenté au paragraphe 3.3.

L'avantage de ce genre de procédé est l'expressivité : la possibilité d'utiliser beaucoup plus facilement la quantification universelle et la grande liberté pour exprimer les problèmes permettent de retranscrire avec une grande précision les problèmes existants et à venir de la littérature concernant les robots mobiles. Cette précision entraîne une plus grande confiance dans les énoncés.

La preuve formelle n'est pas limitée aux instances des protocoles. Elle permet de faire des preuves en quantifiant sur *tous* les protocoles, rendant possible la certification qu'une tâche est *impossible* à réaliser. De plus, la construction de ces assistants à la preuve permet d'y avoir de larges bibliothèques pour les différents modèles, problèmes, protocoles et théorèmes, assurant leur consistance et possibilité de réutilisation.

Cette méthode n'est cependant pas sans défauts. La preuve formelle et l'utilisation des assistants à la preuve nécessitent une grande expertise pour formuler mais aussi pour écrire les preuves. De plus, l'assistant à la preuve requiert interaction et expertise humaine : il y a intrinsèquement un manque d'automatisation par rapport au model-checking.

On pourrait croire ces deux méthodes presque en opposition, l'une fonctionnant avec une logique du premier ordre et avec des quantifications sur des ensembles relativement petits mais de façon totalement automatique, et l'autre où l'écriture « à la main » de preuve permet d'avoir des preuves sur n'importe quel contexte mathématique. En réalité, ces deux modèles sont complémentaires. La figure 3.1 représente leurs champs d'applications possibles.

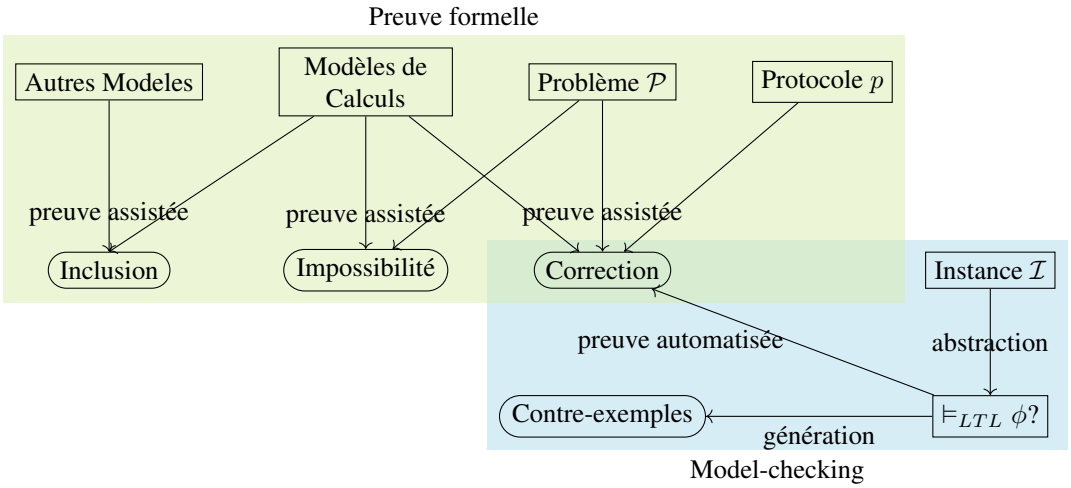


Figure 3.1: Les utilisations de la preuve formelle et du model checking.

termes		types		sorts
3	:	nat	:	Type
Nat.add	:	nat -> nat -> nat	:	Type
1 ?= 3	:	bool	:	Type
eq_refl 0	:	0 = 0	:	Prop.
(O_S) 1	:	0 <> 2	:	Prop.
Nat.le_0_1	:	0 <= 1	:	Prop.
fun n:nat => eq_refl n	:	forall (n:nat), n = n	:	Prop.

valeurs types de donnée preuves propriétés

Figure 3.2: Exemples de types et de valeurs dans Coq. On peut y voir que les preuve et les valeurs sont équivalents dans Coq en terme de typage.

3.3 Coq

Dans cette thèse, nous utiliserons Coq[2] [3], un assistant à la preuve développé par l'Inria et se reposant sur le calcul des constructions, et une théorie des types d'ordre supérieur. Le calcul des constructions (abrégé en *CoC* en anglais) est un λ-calcul avec des types dépendants, dans lequel les types ne sont pas séparés des programmes : ce sont des objets comme les autres. Dans Coq, le calcul des constructions a été étendu avec les constructions inductives [28, 18]. Coq utilise une logique intuitionniste mais qui peut être facilement étendue à la logique classique en admettant l'axiome du tiers exclu. Le principe de Coq utilise la correspondance de Curry-Howard [51] : le typage du langage implique l'existence d'une logique liée à ce langage. De plus, dans Coq, une proposition étant un type de donnée comme un autre, cela permet d'avoir une correspondance entre types et formules, entre programmes et preuves, entre typage de programme et vérification de preuves.

Dans Coq, les propriétés à prouver sont des types de données, et les preuves de ces propriétés sont des valeurs (voir 3.2). Ainsi, la certification par l'assistant qu'une preuve est correcte revient dans Coq à faire une vérification de typage.

La syntaxe du langage de Coq est semblable à celle des langages de programmation fonctionnelle à la ML. Les types des fonctions sont décrits comme habituellement dans les modèles utilisant la curryfication: `A → B → C` représente le type d'une fonction prenant un paramètre `A` et renvoyant une fonction de `B` vers `C` ; cela représente aussi aussi la fonction prenant en paramètre deux objets de type `A` et `B`, et renvoyant un objet de type `C`, pouvant être partiellement appliqué à son premier paramètre.

Pour introduire les termes (types ou programmes), nous utiliserons `Definition`, mais il est également possible de définir de nouveaux types de donnée en listant exhaustivement leurs constructeurs avec `Inductive` (on utilisera alors le filtrage par `match ... with ... end`) ou bien par enregistrement en utilisant `Record` (auquel cas l'accès aux champs se fait par notation pointée). On peut également définir des classes de type (*typeclasses*) en utilisant `Class` (voir figure 3.3a). Les *typeclasses* sont des enregistrements possédant une méthode de recherche automatique.

La majeure partie de ces types[4] peuvent être instanciés avec des valeurs (3.3b).

[2] https://coq.inria.fr/

[3] Cette thèse utilise la version 8.12.0 de Coq

[4] À l'exception notable du cas particulier `False`.

Il est également possible de définir des (sous)-types par intention: `{x: T| P x}` représente le type de tout élément `x` de type `T` couplé à une preuve que `P` est vérifié pour `x`.

Définir une propriété revient en fait à définir un type (figure 3.4). Il reste ensuite à écrire la preuve de cette propriété en utilisant une suite de tactiques ou de preuves. À la conclusion de la preuve, le noyau de coq vérifie que le terme construit possède bien le même type que la propriété de départ. Pour cela, COQ utilise des dérivations de typage, que ça soit par chaînage arrière ou chaînage avant.

Nous pouvons donc définir la notion de `Setoid`, qui est un record regroupant une relation, et une preuve que cette relation est une relation d'équivalence:

```
Record Setoid (A : Type) : Type := Build_Setoid
  { equiv : relation A;  setoid_equiv : Equivalence equiv }
```

Cette notation nous servira tout au long de ce manuscrit pour décrire les relations d'équivalence.

COQ a, à de nombreuses reprises, montré sa puissance, notamment grâce à des résultats importants comme le théorème des quatre couleurs [48], ou le théorème de Feit-Thompson [49] de la théorie des groupes. COQ a permis de développer et de certifier le compilateur *CompCert*[5], un compilateur pour une grande partie du langage C [60]. COQ a aussi été utilisé dans le cadre de l'algorithmique distribuée pour certifier, grâce au framework `PADEC`, une partie non triviale d'un algorithme auto-stabilisant à k sauts construisant un ensemble de *cluster* [3].

[5] http://compcert.inria.fr/index.html

```
(*Une definition de type synonyme. *)
Definition T := bool.

(* Une definition un peu plus complexe formant un produit
cartesien entre deux types. *)
Definition T' := (T*nat).

(* Des types inductifs. *)
Inductive T'' :=
| Cas1: T' → T → T'' (* on sait construire un T'' depuis un T' et un T *)
| Cas2: nat → T''.    (* on sait construire un T'' depuis un nat *)

(* Cas particulier : les listes avec elements de type ''A''. *)
Inductive list (A : Type) :=
| nil : list A                    (* la liste vide est une liste de A *)
| cons : A → list A → list A.    (* on sait construire une liste de A
                                    depuis un A et une liste de A *)

(* Une definition d'une classe de type. *)
Class C2 := {
  champs1 : T;
  champs2 : list bool;
  champs3 : Z → Z;
}.
```

(a) Déclaration de types en COQ.

```
(* Instanciation de ces types. *)
Definition c1 : T := true.

Definition c2 : T' := (c1, 3).

Definition c3 : T'' := (Cas1 (true,2) false).

Definition l1 : list T' := cons c2 (cons (false,8) nil).

Instance i :C2 :=
{| champs1 := true;
   champs2 := nil;
   champs3 := fun x ⇒ (-7); |}.
```

(b) Instanciation de ces types.

Figure 3.3: Code COQ pour introduire les types: la première partie présente quelques déclarations simples de types et la deuxième présente différentes façon d'instancier ces types.

```
(* A   |      B       |              C            | *)
Theorem une_propriete : ∀ (x:nat), x >= 0.
Proof.
intro x. apply le_0_n.    (* D *)
Qed.

Definition une_prop : ∀ x, x >= 0 := le_0_n. (*  E  *)
```

Figure 3.4: Une preuve simple en COQ, avec en (A) le mot clé, en (B) le nom de la propriété, en (C) le corps de la propriété, qui représente son type, et en (D) les applications de tactiques qui permettent de construire la preuve, par chaînage arrière. On peut également donner directement la preuve comme valeur (E).

Chapter 4

Pactole

Dans ce chapitre, nous présentons comment PACTOLE[1] implémente les différentes notions communément présentes pour les algorithmes distribués sur les robots mobiles. Nous allons donc dans un premier temps décrire la représentation des espaces dans PACTOLE. Puis nous présenterons les définitions des robots, ainsi que les façons dont sont décrites leurs limitations. Nous détaillerons ensuite comment sont modélisées les différentes caractéristiques des synchronisations dans PACTOLE. [2]

4.1 L'espace

PACTOLE a pour but de représenter fidèlement mais aussi de manière abstraite les espaces utilisés dans l'algorithmique des robots mobiles.

Dans sa version original, c'est-à-dire avant nos travaux, PACTOLE permet essentiellement la définition d'espaces continus.

On y demande en général de pouvoir décider si deux positions sont égales. Dans le cas des espaces réels, $\mathbb{R}$, $\mathbb{R}^2$, etc., on utilisera les réel axiomatiques de COQ. La plupart des études de cas de PACTOLE utilisent en fait des espaces *vectoriels*.

Pour construire un espace vectoriel, il faut un type de départ, représenté par `T`. Ce type possède une origine, les opérations usuelles que sont l'addition, la multiplication et l'inversion.

Afin de pouvoir travailler sur cet espace, il faut aussi une notion d'équivalence sur le type `T` et la preuve de la décidabilité de cette équivalence, représentées respectivement par `S` et par `H`. Avec `S` et `H`, il est possible maintenant de définir les propriétés de compatibilité d'une fonction avec l'équivalence `S`. Nous pouvons donc définir les propriétés d'associativité et de commutativité de l'addition, ou encore de distributivité de l'addition par la multiplication.

Chacune de ces opérations nécessite aussi une propriété de compatibilité: ces propriétés spécifient que si, lors de deux applications d'une fonction, tous les arguments de chaque applications sont équivalents deux à deux, alors les deux résultats seront équivalents 4.1.

Enfin, pour avoir un espace non dégénéré, l'existence d'une valeur autre que l'origine (`one`) et d'une propriété assurant leur inégalité (`non_trivial`) sont requises.

[1] https://pactole.liris.cnrs.fr/

[2] ces notions sont résumé dans [10]

```
(*     A      |      B     |    C    |   D   |  E  *)
add_compat : Proper (equiv ⩧ equiv ⩧ equiv) add;
```

Figure 4.1: Exemple d'une déclaration de propriété de compatibilité:

A : Le nom de cette propriété.

B : La relation sur première variable de la fonction dont on veut établir la compatibilité.

C : La relation sur la seconde variable de la fonction.

D : La relation que doit vérifier le résultat de la fonction.

E : La fonction qui doit vérifier la propriété de compatibilité.

```
Record RealVectorSpace (T : Type) (S : Setoid T) (H : EqDec S) : Type
  := Build_RealVectorSpace
  { origin : T;
    add : T → T → T;
    mul : R → T → T;
    opp : T → T;
    add_compat : Proper (equiv ⩧ equiv ⩧ equiv) add;
    mul_compat : Proper (eq ⩧ equiv ⩧ equiv) mul;
    opp_compat : Proper (equiv ⩧ equiv) opp;
    add_assoc : ∀ u v w : T, add u (add v w) ≡ add (add u v) w;
    add_comm : ∀ u v : T, add u v ≡ add v u;
    add_origin : ∀ u : T, add u origin ≡ u;
    add_opp : ∀ u : T, add u (opp u) ≡ origin;
    mul_distr_add : ∀ (a : R) (u v : T),
    mul a (add u v) ≡ add (mul a u) (mul a v);
    mul_morph : ∀ (a b : R) (u : T), mul a (mul b u) ≡ mul (a * b) u;
    add_morph : ∀ (a b : R) (u : T), add (mul a u) (mul b u) ≡ mul (a + b) u;
    mul₁ : ∀ u : T, mul 1 u ≡ u;
    one : T;
    non_trivial : one =/= origin }
```

Pour pouvoir faire un espace métrique, nous rajoutons :

```
dist : T → T → R;

dist_defined : ∀ u v, dist u v = 0ℝ ⇔ u ≡ v;
dist_sym : ∀ u v, dist v u = dist u v;
triang_ineq : ∀ u v w, (dist u w ≤ dist u v + dist v w)ℝ
```

Pour créer un espace utilisable par le reste de PACTOLE, il faut l'intégrer dans l'état global du système, représenté par la classe `State`. Cet état requiert un type des positions dans lesquels les robots évoluent, et ce type est créé grâce à la fonction `make_Location`. `State` requiert aussi de connaître le type (`info`) de toutes les informations accessibles aux robots.

```
Class State `{Location} info := {
  get_location : info → location;
  (** States contient une equivalence decidable *)
  state_Setoid :> Setoid info;
```

```
  state_EqDec :> EqDec state_Setoid;
  (** permet d'elever le changement de referentiel d'une position a
  l'etat global, sous certaines conditions *)
  precondition : (location → location) → Type;
  lift : sigT precondition → info → info;
  lift_id : ∀ Pid, lift (existT precondition id Pid) ≡ id;
  get_location_lift : ∀ f state, get_location (lift f state) ≡ projT1 f (get_location state);
  (** et contient des proprietes de compatibilite *)
  get_location_compat :> Proper (equiv ⩧ equiv) get_location;
  lift_compat :>
    Proper ((equiv ⩧ equiv)%signature @@ (@projT1 _ precondition) ⩧ equiv ⩧ equiv) lift }.
```

Ces constructions pour le plan réel existant déjà dans les bibliothèques du prototype, la création de l'espace associé au plan est simplement:

```
Instance Loc : Location := make_Location (R2).
Instance VS : RealVectorSpace location := R2_VS.
Instance ES : EuclideanSpace location := R2_ES.
```

4.2 Les robots

Dans PACTOLE, les robots mobiles sont représentés par un identifiant unique à chaque robot, mais invisible par les robots[3]. Cet identifiant sert uniquement aux preuves.

Nous avons vu qu'il peut exister des robots fautifs, nous séparons donc les robots en deux catégories, les robot fautifs, qui sont appelés byzantins ou `Byz`, et les robots normaux, appelés `Good`. Un robot a ainsi le type `identifier`:

```
  Inductive identifier {G} {B} : Type :=
  | Good : G → identifier
  | Byz : B → identifier.
```

Ici, `G` et `B` sont les ensembles de robots de chaque catégorie qui existe dans le modèle et restent abstraits pour plus de liberté. `B` et `G` sont définis par deux nombres entiers naturels, `nB` et `nG` respectivement, à la création de l'instance des robots:

```
  Notation "'fin' N" := {n : nat | n < N}
  (*...*)

  G := fin nG;
  B := fin nB;
```

Cette définition permet d'avoir des propriétés prédéfinies pour les robots comme l'unicité des robots ou la décidabilité de l'égalité. Cela permet aussi de faire de l'induction sur le nombre de robots. Pour extraire cet entier du type par intention, nous utilisons la fonction `proj1_sig`.

```
proj1_sig =
  fun (A : Type) (P : A → Prop) (e : {x : A | P x}) ⇒ let (a, _) := e in a
  : ∀ (A : Type) (P : A → Prop), {x : A | P x} → A
```

Il faut cependant noter que cela ne représente pas les caractéristiques des robots, cela représente juste l'ensemble des identifiants des robots.

[3]Cela n'empêche pas d'ajouter une forme d'identification perceptible dans l'état visible d'un robot si jamais les hypothèses ne requièrent pas l'anonymat des robots.

4.2.1 Capteurs, perceptions

Dans PACTOLE, une configuration est définie comme une fonction qui étant donné un identifiant de robot, qu'il soit byzantin ou non, renvoie toutes les informations concernant ce robot et en particulier sa position. Ces informations sont stockées dans un type `info`, abstrait.

```
Definition configuration `{State} `{Names} := ident → info.
```

Cela permet de représenter le système à un moment donné. Le système évoluant dans le temps, potentiellement indéfiniment, les exécutions sont simplement des suites infinies de configurations comme dans le paragraphe 2.3.

Cette suite infinie est représentée par un `Stream`. Un `Stream` dans PACTOLE est un objet d'un ensemble défini par coinduction. Il est défini par un ajout (`cons`) en tête, la projection de cet élément de tête (`hd`) et la projection du reste (`tl`) du `Stream`.

```
Section Streams.

Context {A : Type}.
Context `{Setoid A}.

CoInductive t A : Type := cons : A → t A → t A.

Definition hd (s : t A) := match s with | cons e _ ⇒ e end.
Definition tl (s : t A) := match s with | cons _ s' ⇒ s' end.
```

Il est possible de greffer assez simplement sur ces `Stream` la logique temporelle LTL avec des constructions connues dans cette dernière où `P` est une propriété, et `s` un `Stream`:

```
Definition instant (P : A → Prop) := fun s ⇒ P (hd s).
Global Arguments instant P s /.

CoInductive forever (P : t A → Prop) (s : t A) : Prop :=
Always : P s → forever P (tl s) → forever P s.
Global Arguments Always [P] [s] _ _.

Inductive eventually (P : t A → Prop) (s : t A) : Prop :=
| Now : P s → eventually P s
| Later : eventually P (tl s) → eventually P s.
Global Arguments Now [P] [s] _.
Global Arguments Later [P] [s] _.
```

L'espace dans lequel évoluent les robots n'est pas forcément l'espace qu'ils perçoivent. En effet, il a été vu au paragraphe 2.2.1 que les robots pouvaient percevoir une version dégradée de leur espace, par exemple en étant incapables de détecter la multiplicité et ce, dans leur référentiel propre.

Afin d'interdire au protocole l'accès à des informations « privées » au sens où elles débordent du cadre des hypothèses (sur les capteurs par exemple) PACTOLE définit la notion d'*observation*.

Nous voulons rester le plus abstrait possible, nous définissons donc une notion générale de ce type d'observation. Une observation par les robots doit contenir le type de l'espace observé par ces robots (`observation` ci-après), et cet espace doit être décidable. Il est nécessaire aussi d'avoir une fonction de conversion de l'espace dans lequel évoluent les robots vers cet espace d'observation. Nous avons donc une fonction `obs_from_config`, prenant aussi en compte l'état du robot qui observe.

On peut le cas échéant laisser cette fonction abstraite, ce qui ne veut pas dire la laisser sans garanties : il est donc demandé une propriété `obs_from_config_compat` de compatibilité sur cette fonction. Enfin, il nous faut un prédicat `obs_from_config_spect` de sûreté, s'assurant que la fonction de transformation vers l'observation est correcte.

En représentant par `State` un ensemble de propriétés et de fonctions sur l'espace, et par `Names` l'ensemble des (identifiants de) robots :

```
Class Observation {info} `{State info} `{Names} := {
  (** observation est un type decidable *)
  observation : Type;
  observation_Setoid : Setoid observation;
  observation_EqDec : EqDec observation_Setoid;

  (** Conversion d'une configuration (locale) en une observation, etant
  donne l'etat de l'observation *)
  obs_from_config : configuration → info → observation;
  obs_from_config_compat : Proper (equiv ==> equiv ==> equiv) obs_from_config;

  (** Predicat caracterisant une observation correcte pour une
  configuration (locale) *)
  obs_is_ok : observation →  configuration → info → Prop;
  obs_from_config_spec : ∀ config st, obs_is_ok (obs_from_config config st) config st }.
```

PACTOLE fournit aussi des squelettes d'`observation` bien utiles, comme des ensembles ou multiensembles de positions qui peuvent servir à représenter respectivement l'espace visible des positions occupées sans et avec détection de la multiplicité.

Le changement de référentiel se fait en transformant la configuration de base des robots à l'aide d'outils nommés similitudes. Une similitude contient une fonction principale qui est une bijection de l'espace, elle sert à modifier l'origine, l'orientation et la chiralité du robot. Il y a aussi dans la similitude le zoom qui permet de changer l'échelle de l'environnement du robot ainsi qu'une propriété de caractéristique des similitudes concernant la préservation des rapports des distances relatives.

```
Record similarity T `{RealMetricSpace T} := {
  sim_f :> bijection T;
  zoom : R;
  dist_prop : ∀ x y, dist (sim_f x) (sim_f y) = zoom * dist x y}.
```

Nous avons de quoi modéliser les problèmes liés aux capteurs des robots:

- Si le robot est désorienté, nous représentons ça grâce aux similitudes.
- Si le robot a des contraintes sur sa vision comme une portée maximale, il est possible de les modélisé grâce à la fonction `obs_from_config`.
- Si le robot a des restrictions sur la capacité à détecter la multiplicité, il est possible de modéliser ça en définissant le type de l'observation pour représenter ces restrictions.

Par exemple, si nous voulons représenter un modèle où les robots sont anonymes et ont une vision illimitée sans détection de multiplicité, on pourra utiliser l'ensemble de toutes les positions des robots. Si le robot a une distance maximale de vision D et ne détecte pas la multiplicité, la fonction `obs_from_config` ne prendra en compte que les robots dans la sphère de rayon D, un sous-ensemble des positions dans cette sphère pourrait alors constituer une observation. Si en revanche les robots possèdent une vision illimitée mais avec détection de multiplicité, l'`observation` on ira plutôt vers un multi-ensemble de toutes les positions, etc.

Il est important de noter que le type de l'observation étant abstrait, il est tout à fait possible d'avoir un espace observé *différant* de l'espace où les robots évoluent. Nous montrerons au paragraphe 6.3 que ces changements permettent de simplifier des modèles dans lesquels les capteurs des robots sont limités: où l'espace où évoluent les robots est continu mais observé de manière discrète.

4.2.2 Le comportement

La phase *Look* est obtenue par observation.

La phase *Move* est donnée par des changements d'états (incluant la position).

Il reste à définir la phase *Compute* du cycle définissant le comportement des robots à la Suzuki et Yamashita (paragraphe 2.2.2).

Cette phase est réalisée par le protocole distribué. Cette phase calculatoire est donc une fonction qui sur la donnée d'une *observation* retourne un nouvel état courant. Il n'est cependant pas suffisant de se contenter de cette fonction : elle doit vérifier certaines propriétés, comme par exemple que deux observations équivalentes entraînent des prises de décision équivalentes.

La fonction et ses propriétés forment un *robogram*.

`robogram`: un robogram est donc dans PACTOLE un record contenant, d'une part, une fonction principale `pgm` qui, étant donnée l'observation du robot, renvoie les informations nécessaires à la mise à jour du robot, que ça soit ses informations de positionnement ou ses informations autres comme la couleur de sa lumière ou sa mémoire, regroupées dans le type `Trobot` et, d'autre part, les propriétés de compatibilité `pgm_compat` de cette fonction.

```
(* Pour s'assurer que le type de retour du robogram est bien
decidable. *)
Class robot_choice := { robot_choice_Setoid :> Setoid Trobot }.

Record robogram `{robot_choice} := {
  (* le protocole en lui meme *)
  pgm :> observation → Trobot;
  (* les propriete de ce protocole *)
  pgm_compat :> Proper (equiv ==> equiv) pgm}.
```

4.3 Caractéristiques de synchronisation

Il est possible de créer une exécution à l'aide de la fonction `execute` qui part d'une configuration initiale, et qui à l'aide de l'ordonnanceur et du robogram (donc en particulier du protocole) utilise une fonction de transition pour passer d'une configuration à la suivante.

```
Definition execution := Stream.t configuration.

Definition execute (r : robogram) : demon → configuration → execution :=
cofix execute d config :=
Stream.cons config (execute (Stream.tl d) (round r (Stream.hd d) config)).
```

Un démon est lui aussi un `Stream` car à chaque nouvelle configuration, de nouvelles décisions doivent être prises. Nous avons à chaque étape un ensemble de décisions prises par le démon dans ce qui est nommé une `demonic_action`.

Nous appellerons dans la suite *référentiel global* le référentiel adopté par le démon. Il est en général différent des *référentiels locaux* propres à chaque robot (suivant les hypothèses).

Une action démoniaque est un `record` contenant des fonctions en rapport avec l'environnement, tout ce qui n'est pas prévisible par le protocole et des propriétés de sûreté. Ces propriétés et fonctions comprennent les décisions prises par les robots byzantins ainsi que les changements de référentiels et autres informations nécessaires pour les robots corrects.

L'action démoniaque contient enfin la désignation des robots activés.

`activate` : est une fonction qui sert à la représentation d'une exécution SSYNC: si la fonction renvoie `true` pour un robot, alors celui-ci sera activé.

`relocate_byz` : est la fonction qui décrit le comportement des byzantins.

`change_frame` : est une fonction qui, étant donné un robot correct, renvoie des informations servant à la création du référentiel local dudit robot.

`choose_update` : est la fonction qui donne des informations supplémentaires au robot et met à jour ses informations, au besoin.

`choose_inactive` : est la fonction qui met a jour l'état des robots inactifs, au besoin.

Le reste est constitué de propriétés de compatibilité des fonctions ainsi que d'une propriété de sûreté concernant le changement de référentiel.

```
Record demonic_action := {
  activate : ident → bool;
  relocate_byz : configuration → B → info;
  change_frame : configuration → G → Tframe;
  choose_update : configuration → G → Trobot → Tactive;
  choose_inactive : configuration → ident → Tinactive;

  precondition_satisfied : ∀ config g,
                           precondition (frame_choice_bijection (change_frame config g));
  precondition_satisfied_inv : ∀ config g,
  precondition (inverse (frame_choice_bijection (change_frame config g)));

  activate_compat : Proper (Logic.eq ==> equiv) activate;
  relocate_byz_compat : Proper (equiv ==> Logic.eq ==> equiv) relocate_byz;
  change_frame_compat : Proper (equiv ==> Logic.eq ==> equiv) change_frame;
  choose_update_compat : Proper (equiv ==> Logic.eq ==> equiv ==> equiv) choose_update;
  choose_inactive_compat : Proper (equiv ==> equiv ==> equiv) choose_inactive }.
```

`Tframe` : est un type intermédiaire servant à générer la fonction de changement de référentiel `frame_choice_bijection` de la classe `frame_choice`.

`Tactive` : est le type contenant des informations que le démon utilise pour prendre des décisions après qu'un robot a choisi sa destination

`Trobot` : est le type de retour des robogrammes dans PACTOLE

`Tinactive` : est un type similaire à Tactive, mais correspondant aux actions pour les robots non activés.

4.3.1 Round

Le Cœur du modèle de Suzuki et Yamashita est décrit dans la fonction *round* qui permet de faire le passage d'une configuration à la suivante dans l'exécution. Pour modéliser un tel passage, il nous faut dans un premier temps expliquer certaines classes et fonctions.

La classe `frame_choice` est définie lors de la création du modèle, et correspond aux choix du démon concernant l'observation du robot en utilisant le retour du démon dans le type `Tframe` mais aussi les propriétés nécessaires à cette fonction: la propriété de compatibilité et celle d'existence de l'équivalence sur `Tframe`.

La classe `update_choice` contient juste des propriétés sur le type `Tactive`.

De même pour la classe `inactive_choice` concernant le type `Tinactive`.

La classe `inactive_function` sert à définir une fonction qui permet de mettre à jour des informations pour un robot inactif, en utilisant `Tinactive`.

la classe `update_function` est quant à elle la classe contenant la fonction permettant la mise à jour des informations d'un robots et la propriété de compatibilité de cette fonction.

```
Class frame_choice := {
  frame_choice_bijection :> Tframe → bijection location;
  frame_choice_Setoid :> Setoid Tframe;
  frame_choice_bijection_compat :> Proper (equiv ==> equiv) frame_choice_bijection }.
Global Existing Instance frame_choice_bijection_compat.

Class update_choice := {
  update_choice_Setoid :> Setoid Tactive;
  update_choice_EqDec :> EqDec update_choice_Setoid }.

Class inactive_choice := {
  inactive_choice_Setoid :> Setoid Tinactive;
  inactive_choice_EqDec :> EqDec inactive_choice_Setoid }.

Class update_function `{robot_choice} `{frame_choice} `{update_choice} := {
  update :> configuration → G → Tframe → Trobot → Tactive → info;
  update_compat :> Proper (equiv ==> Logic.eq ==> equiv ==> equiv ==> equiv ==> equiv) update

Class inactive_function  `{inactive_choice} := {
  inactive :> configuration → ident → Tinactive → info;
  inactive_compat :> Proper (equiv ==> Logic.eq ==> equiv ==> equiv) inactive }.
```

Grâce à ces fonctions, il est possible de se placer dans le référentiel du robot, pour pouvoir appliquer le protocole dans le référentiel du robot, et après repasser dans le référentiel global, et modifier les informations du robot. Pour passer de fonctions mettant à jour les informations de robots particuliers à une mise à jour de la configuration en général, nous utilisons la fonction `round`.

Cette fonction est le cœur même du modèle dans PACTOLE*, elle caractérise le modèle de Suzuki et Yamashita.*

Cette fonction prend en paramètre le robogramme des robots, l'étape du démon correspondant et la configuration courante, et renvoie la configuration suivante de l'exécution.

```
Definition round (r:robogram) (da:demonic_action)
                 (config:configuration) : configuration :=
  fun id ⇒
```

Une configuration étant une fonction des identifiants vers les informations des robots, la fonction `round` fait une séparation de cas en fonction des robots. Pour chaque robot, elle regarde si le démon l'active ou non grâce à la fonction `activate`.

```
let state := config id in
if da.(activate) id
```

Si un robot n'est pas activé, alors la fonction `round` fait appel à la fonction `inactive`, changeant les informations de ces robots.

```
inactive config id (da.(choose_inactive) config id).
```

Si un robot est activé, s'il était byzantin, la valeur retournée est le résultat décidé par le démon.

```
match id with
| Byz b ⇒ da.(relocate_byz) config b
```

Sinon, il faut changer de référentiel pour arriver à la configuration locale du robot `local_config`, c'est-à-dire la configuration dans le référentiel du robot.

```
  | Good g ⇒
  let frame_choice := da.(change_frame) config g in
  let new_frame := frame_choice_bijection frame_choice in
  let local_config := map_config (lift (existT precondition new_frame
  (precondition_satisfied da config g)))
  config in
```

On détermine la perception du robot grâce à l'observation `obs`. C'est la phase *Look* du cycle déterminant le comportement des robots mobiles.

```
  let local_pos := get_location (local_config (Good g)) in
  let obs := obs_from_config local_config local_pos in
```

À partir de cette observation le robogramme retourne un résultat `local_robot_decision` correspondant aux prises de décisions. C'est la phase *Compute*.

```
  let local_robot_decision := r obs in
(* le demon choisit comment realiser le changement d'etat *)
  let choice := da.(choose_update) local_config g local_robot_decision in
```

Avec toute ces informations, la fonction `update` met en place les nouvelles informations concernant la position et l'état du robot. C'est donc l'action *Move*.

```
  (* le veritable changement de l'etat des robots est realise par la
  fonction [update] *)
  let new_local_state := update local_config g frame_choice local_robot_decision choice in
```

Il ne reste plus qu'à inverser les changements fait sur la vision du robot pour revenir au référentiel global pour traduire ça dans l'espace des configurations.

```
  (* retour au referentiel global *)
  lift (existT precondition (inverse new_frame)
                            (precondition_satisfied_inv da config g)) new_local_state
```

Le code entier de la fonction `round` est à la figure 4.2
Nous avons donc une implémentation de la notion de *round* de Suzuki et Yamashita dans PACTOLE.

4.3.2 Propriétés des démons

Avec toutes les fonctions précédentes, il est possible de définir des propriétés plus avancés sur les démons dans PACTOLE pour retranscrire fidèlement les modèles de la littérature. Par exemple, un démon FSYNC dans PACTOLE est défini par le fait que la fonction `activate` renvoie `true` pour chaque robot. De même, un démon SSYNC est un démon pour lequel il est possible que des robots soient laissés inactifs, dans ce cas là, ces robot ne font rien.

```
  Definition FSYNC_da da := ∀ id, da.(activate) id = true.

  Definition FSYNC d := Stream.forever (Stream.instant FSYNC_da) d.

  Definition SSYNC_da da := ∀ id, da.(activate) id = false →
  ∀ config, inactive config id (da.(choose_inactive) config id) ≡ config id.

  Definition SSYNC d := Stream.forever (Stream.instant SSYNC_da) d.
```

```
Definition round (r:robogram) (da:demonic_action) (config:configuration)
           : configuration :=
fun id ⇒
let state := config id in
if da.(activate) id
then
  match id with
  | Byz b ⇒ da.(relocate_byz) config b
  | Good g ⇒
  let frame_choice := da.(change_frame) config g in
  let new_frame := frame_choice_bijection frame_choice in
  let local_config := map_config (lift (existT precondition new_frame
                                        (precondition_satisfied da config g)))
                                 config in
  let local_pos := get_location (local_config (Good g)) in
  let obs := obs_from_config local_config local_pos in
  let local_robot_decision := r obs in
  (* le demon choisit comment realiser le changement d'etat *)
  let choice := da.(choose_update) local_config g local_robot_decision in
  (* le veritable changement de l'etat des robots est realise par la
  fonction [update] *)
  let new_local_state := update local_config g frame_choice
                                local_robot_decision choice in
  (* retour au referentiel global *)
  lift (existT precondition (inverse new_frame)
  (precondition_satisfied_inv da config g)) new_local_state
  end
else inactive config id (da.(choose_inactive) config id).
```

Figure 4.2: Le code de la fonction `round`

D'autres propriétés comme l'équité d'un démon sont présentes aussi dans PACTOLE.

La proposition `LocallyFairForOne r d` spécifie que le démon `d` activera le robot `r` dans un nombre fini de rounds, que ce soit maintenant (`Now`) ou plus tard (`Later`). La proposition `Fair d` requiert que `LocallyfairForOne` soit toujours vérifié pour tout robot.

```
Inductive LocallyFairForOne id (d : demon) : Prop :=
| NowFair : activate (Stream.hd d) id ≡ true → LocallyFairForOne id d
| LaterFair : activate (Stream.hd d) id = false →
LocallyFairForOne id (Stream.tl d) → LocallyFairForOne id d.

Definition Fair : demon → Prop :=
  Stream.forever (fun d ⇒ ∀ id, LocallyFairForOne id d).
```

4.4 Quelques résultats

Un certain nombre de résultats formels ont été obtenus grâce à PACTOLE comme, par exemple, des certifications d'algorithmes de rassemblement dans $\mathbb{R}$ [30] ou $\mathbb{R}^2$ [32] où il est décrit un espace où les robots évoluent et où il est proposé un algorithme en définissant la fonction `pgm` dans le `robogram`. De là, les auteurs prouvent formellement qu'avec n'importe quel démon, l'algorithme résout le problème du rassemblement.

D'autre preuves ont été faites grâce à PACTOLE, notamment sur l'impossibilité d'avoir la certitude du rassemblement dans $\mathbb{R}$ si le nombre de robots est pair, qu'importe le protocole utilisé [31], l'impossibilité d'obtenir la convergence avec une certaine *proportion*[4] de robots byzantins [6], ou encore certaines solutions pour le rassemblement même si les robots ne perçoivent pas la multiplicité [11].

PACTOLE est disponible depuis `https://pactole.liris.cnrs.fr`.

[4] Il s'agit bien d'une proportion et non pas d'un nombre fixé.

Chapter 5

Graphes comme espaces dans PACTOLE

Les espaces discrets sont largement présents dans la littérature concernant les robots autonomes. Le prototype original de PACTOLE ne permettant pas de travailler sur ce genre d'espace, il était important d'y intégrer les moyens d'exprimer ces espaces discrets de la façon la plus simple possible.

Nous proposons une interface générale pour définir des graphes en adéquation avec le cadre formel. Nous n'avons pas la prétention de réaliser une bibliothèque du calibre de *loco* [25], une bibliothèque permettant de manipuler des graphes arbitraires. Nous donnons davantage les moyens de définir des espaces se comportant comme des graphes plutôt que des graphes pour eux-mêmes. De plus, un des buts de PACTOLE étant de faciliter la specification à des utilisateurs extérieurs au milieu de la preuve formelle nous veillons autant que faire se peut à la simplicité de création de ces espaces, y compris pour les non-experts.

5.1 Modèle général des graphes

Pour construire un espace, il nous faut un ensemble décidable de positions. Pour cela, nous définissons une classe `Graph` qui représente les graphes en général. Prenons le principe de base de la création de graphes: deux ensembles, un pour les sommets (`V`) et un pour les arêtes (`E`). Comme nous voulons que ces deux types puissent être utilisés comme positions des robots, ces types possèdent une notion d'équivalence et de décidabilité.

Les arêtes allant toujours d'un sommet vers un autre, et comme nous voulons garder le type des arêtes abstrait, nous rajoutons 2 fonctions : source (`src`) et target (`tgt`), servant à relier le type `E` au type `V` et retournant respectivement le point de départ et le point d'arrivée d'une arête.

Nous avons des fonctions de l'ensemble des arêtes vers l'ensemble des sommets. Il nous faut aussi une fonction dans l'autre sens: la fonction `find_edge` est une fonction qui, étant donnés deux sommets, renvoie un type option (`None`/`Some` x) pour indiquer s'il existe une arête entre les deux sommets. [1]

La cohérence entre les fonctions `src` et `tgt`, d'une part, et la fonction `find_edge`, d'autre part, est garantie par la propriété `find2st`. Cette fonction vérifie que s'il existe une arête entre deux sommets, alors les fonctions `src` et `tgt` rendent bien ces sommets.

Toutes ces fonctions représentent la topologie dans laquelle évoluent les robots, mais ces robots ne perçoivent pas forcément exactement cet espace. Comme il a été vu précédemment dans la section 4,

[1] `None` représente le fait qu'il n'y a pas d'arête entre le premier et le second sommet et `Some` E représente le fait qu'il existe l'arête E partant du premier sommet et allant au deuxième.

notre modèle nécessite aussi une possibilité de changer de référentiel. Nous avons donc ajouter un type de bijection pour les graphes, les isomorphismes de graphes, contenant une bijection pour chaque type, sommet et arête, ainsi que des propriétés s'assurant que l'organisation du graphe ne change pas. Par exemple dans un anneau, un isomorphisme est une rotation de l'anneau, ou une symétrie axiale.

```
Record isomorphism := {
  iso_V :> bijection V;
  iso_E : bijection E;
  iso_morphism : ∀ e, iso_V (src e) ≡ src (iso_E e)
                   ∧ iso_V (tgt e) ≡ tgt (iso_E e);
}.

Definition bij_trans_V (c : Z) : @Bijection.bijection ring_node
   (@V_Setoid _ _ localRing) :=
refine {|
  Bijection.section := fun x ⇒ of_Z (to_Z x - c);
  Bijection.retraction := fun x ⇒ of_Z (to_Z x + c) |}.
(* ... *)
Definition bij_trans_E (c : Z) : @Bijection.bijection ring_edge (@E_Setoid _ _ localRing).
refine {|
  Bijection.section := fun x ⇒ (of_Z (to_Z (fst x) - c), snd x);
  Bijection.retraction := fun x ⇒ (of_Z (to_Z (fst x) + c), snd x) |}.
(*  ... *)
Definition trans (c : Z) : isomorphism localRing.
refine {|
  iso_V := bij_trans_V c;
  iso_E := bij_trans_E c |}.
```

5.2 Exemples de graphes particuliers

Pour montrer l'expressivité et les possibilités de ce modèle de graphe, nous avons aussi décrit dans PACTOLE un modèle de graphe plus particulier: l'anneau (au sens graphe du terme). Un anneau est un graphe connexe où chaque sommet est relié à deux et seulement deux autres sommets.

Pour pouvoir retranscrire ce modèle dans PACTOLE, nous avons numéroté les n sommets du graphe de 0 à $n-1$, faisant de `V` un ensemble d'entiers relatifs. Les arêtes y sont une paire contenant le sommet de départ de l'arête et le sens dans laquelle l'arête se dirige, défini par trois valeurs: `Forward` quand la valeur entre l'arête de départ et celle d'arrivée augmente, `Backward` quand cette valeur décroît. Nous contournons le cas particulier des arêtes entre 0 et $n-1$ en calculant toute ces opérations modulo n.

COQ nécessitant de vérifier tous les cas, les arêtes bouclant sur un sommet sont définies, ces arêtes utilisent comme indication de direction la troisième valeur : `AutoLoop`.

Les définitions de `src` et `tgt` sont définies simplement, tout comme la fonction de recherche d'arête :

```
Definition origin := 0 mod n.

Definition add (x y : Z) : Z :=  (Zmod (x + y) n).

Definition mul (x y : Z): Z :=  (Zmod (Z.mul x y) n).

Definition eq (x y : Z): Prop := Zmod x n = Zmod y n.

Definition opp (x: Z): Z := (n - x) mod n.

Definition dist (x y : Z) : Z := Z.min (add (opp y) x) (add (opp x) y).
```

Figure 5.1: Code des opérations pour décrire un anneau.

```
Lemma triang_ineq : ∀ x y z, dist x z ≤ add (dist x y) (dist y z).
```

Figure 5.2: Inégalité triangulaire dans un anneau.

```
Definition src (e:E) := fst e.
Definition tgt (e:E) :=
  match snd e with
  | Forward ⇒ (fst e) + 1
  | Backward ⇒ (fst e) - 1
  | AutoLoop ⇒ fst e
  end.

(* ici Loc.add est une fonction s'assurant de la bonne
   correspondance des types entre les fonctions *)
Definition find_edge v1 v2 :=
  if eq_dec v1 (Loc.add v2 1)
  then Some (v1, Backward)
  else if eq_dec (Loc.add v1 1) v2
  then Some (v1, Forward)
  else if eq_dec v1 v2
  then Some (v1, AutoLoop)
  else None.
```

Nous introduisons l'espace associé à l'anneau dans PACTOLE comme suit: nous définissons les opérations comme `add` et `opp` grâce au modulo, c'est-à-dire le reste de la division euclidienne (voir figure 5.1). En effet, bouger d'un sommet x à un autre revient à faire $(x+1) mod\ n$ ou $(x-1) mod\ n$ suivant le sens, avec n le nombre total de sommets du graphe.

La définition d'un espace nécessite aussi de prouver les propriétés nécessaires pour la création d'espace dans PACTOLE comme par exemple l'inégalité triangulaire. Cette propriété est définie de façon simple comme montré à la figure 5.2 .

Il est possible de voir dans la figure 5.3 comment cette propriété à première vue simple est fastidieuse à prouver en COQ. Ce dépliage nous donne 27 possibilités, dont un grand nombre sont triviales et résolues grâce à une application de `omega`, la tactique de COQ résolvant les problèmes simples dans $\mathbb{Z}$.

Il nous reste 4 cas non triviaux, mais qui se regroupent en deux catégories contenant des cas équivalents au sens de l'anneau près.

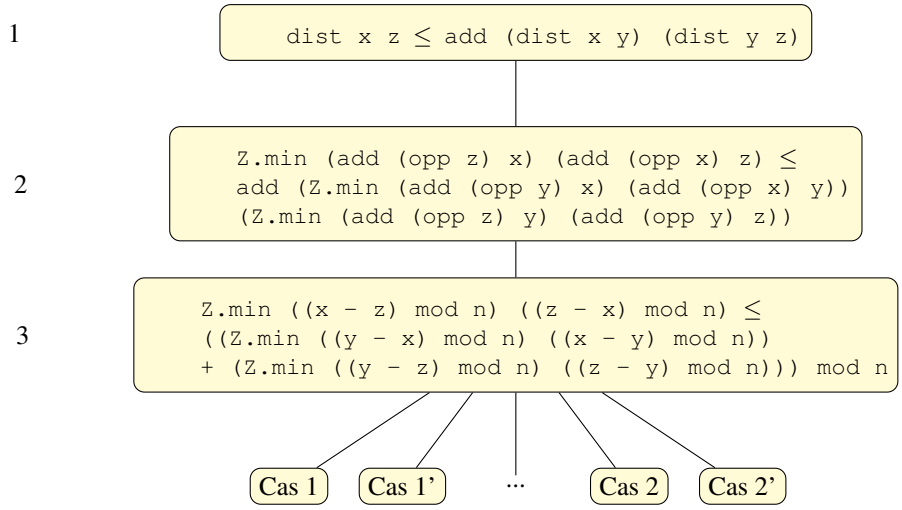


Figure 5.3: Pour prouver cette inégalité en Coq, nous allons commencer par développer les définitions de `dist`, pour pouvoir voir tous les différents cas auxquels nous allons devoir faire face (1).
Puis nous allons déplier les définitions des opérations modulo n, pour pouvoir plus facilement travailler avec le scope `Z` de Coq, un scope étant un environnement de Coq pour que les opérations soient en priorité regardées sur des valeurs de type voulu, ici `Z` (2).
Nous allons ensuite différentier toutes les possibilités des termes de la forme `Z.min ((a − b) mod n) ((b − a) mod n)` en distinguant trois cas : le cas où les deux parties sont égales et chacun des cas où une des partie est strictement plus petite que l'autre (3).

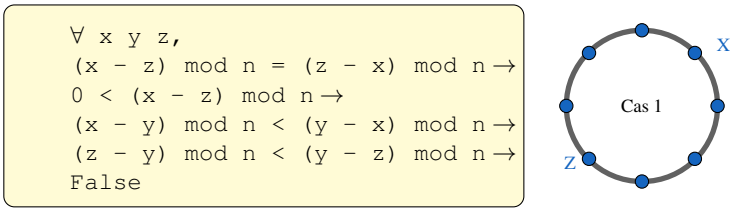


Figure 5.4: Le premier des cas non triviaux concernant la preuve de l'inégalité triangulaire dans les graphes

```
∀ x y z,
(x − z) mod n = (z − x) mod n →
0 < (x − z) mod n →
(y − x) mod n < (x − y) mod n →
(y − z) mod n < (z − y) mod n →
False
```

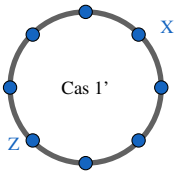


Figure 5.5: Un cas similaire au premier cas de la preuve.

```
∀ x y z,
(x − z) mod n < (z − x) mod n →
0 < (x − z) mod n →
(x − y) mod n < (y − x) mod n →
(z − y) mod n < (y − z) mod n →
(x − z) mod n ≤ ((x − y) mod n + (z − y) mod n) mod n.
```

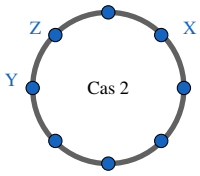


Figure 5.6: Le deuxième cas de la preuve d'inégalité triangulaire dans les graphes, avec la formule en Coq, et sa représentation.

Dans le premier cas (5.4), nous sommes dans un des cas où X et Z sont diamétralement opposés sur l'anneau, ce qui est exprimé par les deux premières lignes. Le reste de l'énoncé nous donnes les hypothèses permettant de prouver la contradiction car nous sommes dans un cas où Y est plus proche de la position de X dans un sens que dans l'autre mais est plus proche de Z dans le même sens, ce qui est impossible.

En dépliant la définition de `dist`, nous retrouvons la propriété voulue. De là, il reste beaucoup de travail arithmétique pour enlever les modulos afin que la tactique `omega` puisse établir l'impossibilité de ces inégalités. Le cas similaire 5.5 est différent car le robot Y est plus proche dans le sens inverse, c'est-à-dire que la comparaison des valeurs de `(x − y) mod n` et `(y − x) mod n` donne le résultat inverse.

Dans les autres cas non triviaux, regardons en premier le cas 5.6 où il nous faut prouver que le robot Y est plus proche de X dans un sens, et plus proche de Z dans le même sens, sachant que Z est entre Y et X. Pour cette preuve nous montrons en premier lieu que les trois valeurs `(x−z)mod n`, `(x−y) mod n` et `(z−y)mod n` sont toutes inférieures ou égales à $n/2$, puis nous nous en servons pour retirer le modulo du membre droit de l'inéquation finale ce qui mène à `(x − z) mod n ≤(x − y) mod n + (z − y) mod n`. Nous faisons quelques opérations sur les modulos pour arriver à l'inéquation `(x − z) mod n ≤(x − y) mod n + (z − y) mod n`. De là, nous transformons `(x−y) mod n` en `(x − z) mod n + (z − y) mod n`, ce qui résout l'inéquation.

Et ici encore, il y a le cas similaire au cas 2, mais où les valeurs des comparaisons sont inversées. Dans ce cas, la position de Y est plus proche de X que de Z (5.7).

Ces preuves totalisent un millier de lignes pour prouver l'existence de l'espace associé au graphe.

```
∀ x y z,
(x - z) mod n < (z - x) mod n →
0 < (x - z) mod n →
(y - x) mod n < (x - y) mod n →
(y - z) mod n < (z - y) mod n →
(x - z) mod n ≤ ((y - x) mod n + (y - z) mod n) mod n
```

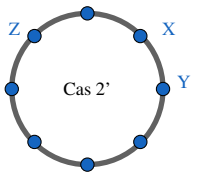


Figure 5.7: Le cas similaire au cas 2 de la preuve de l'inégalité triangulaire.

5.3 Graphes en action : une preuve d'impossibilité

Pour illustrer l'implantation des graphes dans PACTOLE, nous en présentons une utilisation en établissant formellement un résultat d'impossibilité concernant l'exploration avec stop dans un anneau.

L'exploration avec stop est satisfaite par un protocole s'il vérifie les deux propriétés suivantes:

1. Chaque sommet du graphe est visité en temps fini par au moins un robot.
2. Une fois tous les nœuds visités, tous les robots sont arrêtés pour toujours.

Nous montrons formellement que si nous avons un anneau de taille n, et un nombre de robots k dans cet anneau, si k divise n, alors aucun protocole ne résout l'exploration avec stop en FSYNC.

5.3.1 Exploration avec stop

Prouver qu'un protocole donné ne résout pas l'exploration avec stop à l'aide d'un contre-exemple revient à donner une configuration initiale et un démon rendant au moins une des deux conditions nécessaires à l'exploration avec stop fausse. Comme il n'est pas possible d'énumérer tous les protocoles pour un nombre arbitraire k de robots sur un graphe de taille arbitraire n, prouver qu'aucun protocole ne peut résoudre cette exploration requiert de classer tous les protocoles dans un ensemble fini de classes, et de fournir des contres-exemples génériques à chaque classe.

Il nous faut présenter une configuration initiale et un démon admissibles tels que pour toute classe de protocoles, nous pouvons prouver que:

- soit le protocole s'arrête (au sens où les robots restent immobiles), mais tous les sommets du graphes n'ont pas été visités,
- ou il existe des robots se déplaçant sur le graphe, mais il nous est possible de prouver que la configuration qui en découle est suffisamment similaire à la configuration initiale pour certifier que les robots ne s'arrêteront jamais.

Nous utilisons la condition classique sur la configuration initiale qu'il n'existe pas deux robots sur le même sommet du graphe.

Nous ne considérons que des configurations où k est plus petit que n sinon soit la condition sur les configurations initiales est fausse, soit la résolution est triviale car il suffit de mettre un robot sur chaque sommet du graphe.

Pour que notre contre-exemple s'applique à la fois aux modèles FSYNC et SSYNC, nous choisirons préférentiellement un démon FSYNC. Comme un tel démon peut aussi bien être considéré comme un démon semi-synchrone (qui active tous les robots à chaque nouveau round), la même preuve couvrira les deux modèles d'exécutions. Une synchronisation FSYNC peut aussi être simulée avec un démon ASYNC, ce qui rend la preuve valide avec tous les modèles de synchronisation. Attention toutefois : nous n'avons pas encore développé avec PACTOLE la preuve que le démon ASYNC peut représenter un démon FSYNC ou SSYNC.

5.3.2 Argument de preuve

Nous nous plaçons dans un anneau discret et fini, nous y considérons des robots oublieux et anonymes, avec une détection de la multiplicité, en synchronisation FSYNC. Nous supposons qu'il n'y a aucun robot byzantin: si l'exploration avec stop est impossible malgré l'absence de robots byzantins, elle est certainement impossible avec également : il suffit qu'ils se comportent comme des robots corrects. Les sommets de l'anneau sont également anonymes, et il n'y a pas de sommet spécifiquement connu par les robots comme un point de référence.

Les mouvements d'un robot sont rigides et limités à trois possibilités:

- Soit le robot bouge vers son voisin dans le sens horaire,
- Soit il bouge vers son voisin dans le sens anti-horaire,
- Soit il ne bouge pas.

L'argument de la preuve de l'impossibilité de l'exploration avec stop est découpé en deux cas:

1. Il y a strictement moins de deux robots,
2. Il y a au moins deux robots et leur nombre divise la taille de l'anneau.

Trop peu de robots

Nous commençons par une remarque très simple mais très importante: à cause de la restriction sur le nombre de robots k qui est inférieur à la taille du graphe n, un protocole qui s'arrête dans une configuration analogue à une configuration initiale ne peut pas résoudre l'exploration avec stop.

Cela implique que, étant dans une configuration qui pourrait être une configuration initiale, un protocole résolvant l'exploration avec stop doit faire au moins un mouvement. Pour rappel, dans notre cas, les configurations initiales acceptées correspondent à n'importe laquelle des configurations où il n'existe pas deux robots qui sont sur un même noeud.

Un protocole résolvant l'exploration *avec stop* doit arriver à une configuration qui n'est pas acceptée comme configuration initiale, c'est-à-dire une configuration avec deux robots sur le même sommet.

Une conséquence de cette remarque est qu'il n'existe aucun protocole résolvant l'exploration avec stop avec un unique robot.

Le nombre de robots divise la taille de l'anneau

S'il y a au moins deux robots, et si le nombre de robots divise la taille de l'anneau, alors il est possible de créer une configuration initiale symétrique assurant que tous les robots ont exactement le même comportement.

La configuration globale reste ainsi dans un état où tous les robots ont exactement la même vision de leur environnement et donc choisissent le même mouvement si ils sont activés. Comme l'exploration avec stop est résolue seulement par un protocole qui réussi pour tous les démons et toutes les désorientations possibles, il suffit de trouver un démon qui fait échouer tous les protocoles avec cette configuration initiale. Nous choisissons le démon qui active tous les robots à chaque round et leur donne tous la même orientation.

Ainsi, tout protocole aura donc un des comportements suivants:

- Soit le protocole ordonne a chaque robot de ne pas bouger, ce qui signifie que le protocole stoppe comme à la figure 5.8,

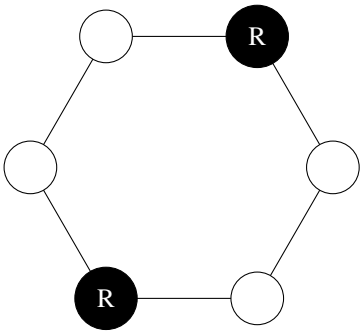


Figure 5.8: Dans le cas d'un protocole faisant s'arrêter les robots, nous nous retrouvons ici avec deux sommets visités, mais les quatre autres ne le sont pas et ne le serons jamais, car on reste dans la même configuration.

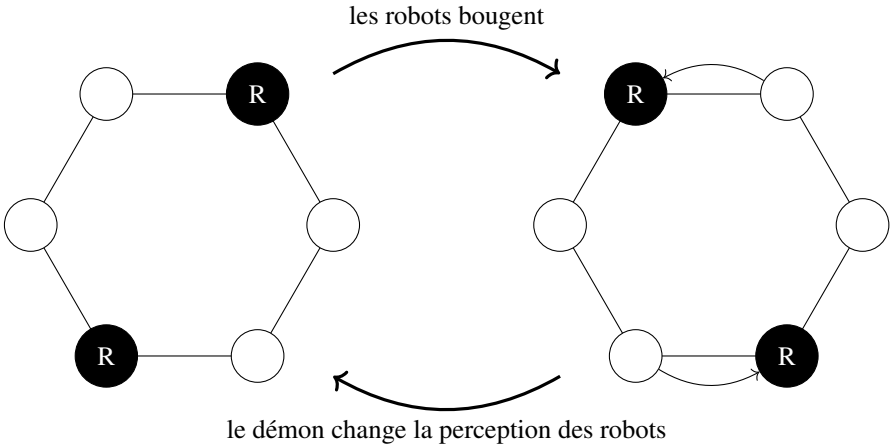


Figure 5.9: Si le protocole ordonne à chaque robot de bouger dans la même direction (ici vers la gauche), le démon changera la perception des robots lors de la prochaine configuration pour qu'elle soit indistinguable de l'initiale, et donc que les robots seront en mouvement perpétuel.

- Soit les robots bougent indéfiniment et on se retrouve avec une suite d'actions comme à la figure 5.9.

De là, nous déduisons qu'il n'existe pas de protocole pouvant résoudre l'exploration avec stop dans un anneau si le nombre de robots divise la taille de l'anneau.

5.3.3 Formalisation de l'impossibilité de l'exploration avec stop

Contexte

Avec des robots anonymes, l'observation ne doit contenir aucune information concernant d'éventuels noms. Avec la détection de la multiplicité, un multi-ensemble contenant les positions occupées est une `observation` acceptable (c'est-à-dire un multi-ensemble avec un élément distinct dénotant la position du robot considéré).

Les robots sont oublieux, par conséquent il leur est associé un nouveau référentiel à chaque activation. Dans le cas d'un espace basé sur un graphe, un isomorphisme (de graphe) est appliqué au graphe avant d'extraire l'observation pour retirer toute information se reposant sur les identifiants des sommets.

Comme nous utilisons un démon FSYNC, la fonction `activate` de l'action démoniaque renverra toujours `true`. la fonction permettant de calculer l'observation sera quant à elle créée à partir du paramètre `change_frame`.

Le problème

Le problème de l'exploration avec stop requiert que les robots visitent collectivement tous les sommets du graphes et s'arrêtent une fois l'exploration complétée. Pour formaliser cette spécification, nous utilisons le prédicat `Will_be_visited` tel que, si `v` est un sommet, et `exe` est une exécution, la propriété `Will_be_visited v exe` est vérifiée si et seulement si au moins un robot visite le sommet `v` durant au moins un round de `exe`:

```
(* Nous utilisons les operations de la logique temporelle classique.
   on peut lire la definition cidessous comme: ''A un moment de
   l'execution, le sommet v sera visite.'' *)
Definition Will_be_visited v exc :=
  Stream.eventually (Visited_now v) exc.
```

Le prédicat `Visited_now` est vérifié pour toute execution `exe` pour laquelle la configuration de tête a un robot au sommet `v`. Nous caractérisons une exécution qui termine par l'existence d'un round a partir duquel l'exécution est arrêtée (*stopped*), c'est-à-dire qui ne bouge pas (*stalls* pour toujours). C'est facilement atteint en combinant les opérateurs temporels usuels:

```
(* Une execution est ''stalled'' si la configuration courante est la
meme que la configuration suivante.*)
Definition Stall (exc : execution) :=
  Config.eq (hd exc) (hd (tl exc)).

  (* Une execution est arretee si elle est ''stalled'' pour toujours  *)
Definition Stopped (exc : execution) :=
  Stream.forever Stall exc.

  (* Une execution s'arretera si a un moment accessible dans le futur,
  elle devient arretee.  *)
Definition Will_stop (exc : execution) :=
  Stream.eventually Stopped exc.
```

Une solution à l'exploration avec stop est un robogramme tel que, pour toute configuration initiale admissible et pour tout démon admissible, l'exécution du robogramme satisfait les propriétés précédentes.

Nous avons donc la définition de satisfaction de l'exploration avec stop suivante:

```
Definition FullSolExplorationStop (r : robogram) (d : demon) :=
(* L'exploration avec stop est resolue par le robogramme r pour le demon d si: *)
∀ config,
(* pour tout configuration de depart *)
(∀ l, Will_be_visited l (execute r d config))
(* toute position sera visite lors de l'execution *)
∧ Will_stop (execute r d config).
(* et cette execution s'arretera. *)
```

Nous avons précédemment mentionné que nous considérons une configuration initiale `c` comme étant admissible, ou *valide*, dès qu'elle ne contient aucune tour, c'est-à-dire dès qu'elle ne contient pas deux robots sur le même sommet. Le prédicat correspondant est nommé `Valid_starting_conf` dans le développement Coq:

```
  (* La fonction config est une fonction injective des identifiants
  vers les etats des robots *)
  Definition Valid_starting_config config : Prop :=
  Util.Preliminary.injective (@Logic.eq ident) (@equiv _ state_Setoid) config.
```

Pour avoir un prédicat seulement sur les robogrammes, nous considérerons uniquement des démons équitables.

Pour finir, il est possible d'écrire le prédicat caractérisant les robogrammes solutions de l'exploration avec stop:

```
Definition Explore_and_stop (r : robogram) :=
(* Un robogramme resout l'exploration avec stop si *)
∀ (c : configuration) (d : demon),
(* pour toute configuration ''c'' et tout demon ''d'' *)
Valid_starting_conf c →
(* sachant que ''c'' est une configuration valide de depart *)
Fair d →
(* et que ''d'' est un demon equitable *)
(∀ l, Will_be_visited l (execute r d c))
(* toute position sera visite lors de l'execution *)
∧ Will_stop (execute r d c).
(* et l'execution s'arretera. *)
```

Ainsi nous avons notre théorème: si la taille de l'anneau et le nombre de robots respectent nos hypothèses, il n'est pas possible de réaliser l'exploration avec stop qu'importe le robogramme.

```
  Hypothesis kdn : (ring_size mod kG = 0)%N.
  Hypothesis k_sup1 : (1 < kG)%N.
  Hypothesis k_inf_n : (kG < ring_size)%N.
  Variable r : robogram.
  (* ... *)
  Theorem no_exploration : ¬ Explore_and_Stop r.
```

Cas 1: Les tours sont requises

Ce cas d'étude consiste à établir que l'exploration avec stop est impossible à réaliser quand il existe moins de deux robots. Le théorème à prouver est:

```
(* Pour realiser l'exploration avec stop, il est necessaire d'avoir
plus d'un robot *)
Theorem no_exploration_k_inf₂ :
  ∀ r, Explores_and_stops r → kG > 1.
```

Pour cela, on commence par retirer l'hypothèse `k_sup₁`, pour ensuite prouver notre théorème. Le lemme le plus important dans cette preuve exprime le fait qu'un robogramme résolvant l'exploration avec stop ne peut pas s'arrêter dans une configuration qui pourrait être une configuration admissible comme initiale:

```
Lemma no_stop_on_starting_conf : ∀ r c d,
  (* Pour tout robogram r, configuration c et demon d,  *)
  Explores_and_stops r →
  (* si r resout l'exploration avec stop, *)
  Valid_starting_conf c →
  (* que c est une configuration de depart valide, *)
  Fair d →
  (* et que d est un demon equitable, *)
  ¬ Stopped (execute r c d).
  (* alors l'execution partant de r et utilisant le demon d
  n'est pas arrete a la configuration c. *)
```

En effet, Si nous présupposons qu'un robogramme `r` s'arrête dans une configuration initiale valide `c` quand il est orchestré par un démon équitable `d`, alors il est possible de prendre la configuration `c` et le démon `d` comme contre-exemple du fait que `r` résout l'exploration avec stop car il est impossible de finir l'exploration avec stop sur une configuration initiale valide, cette configuration pouvant être le point de départ d'une autre exécution.

Nous avons donc le théorème suivant nécessitant 200 lignes de code pour prouver l'impossibilité si il existe moins de 2 robots:

```
no_exploration_k_inf₂ : ∀ r d config,
Fair d →
Explore_and_Stop r →
Valid_starting_config config →
(kG > 1)ℕ.
```

Cas 2: k divise n

Ce cas consiste à établir que l'exploration avec stop est impossible à réaliser quand le nombre de robots k divise la taille de l'anneau n. Le théorème principal à prouver peut être directement défini:

```
Theorem no_exploration_k_divides_n :
(* si n est la taille de l'anneau et k le nombre de robots *)
(n mod k) = 0 →
∃ d, FSYNC d ∧
(* si la division euclidienne de n par k donne un reste de 0 *)
∀ r, ¬ (Explores_and_stops r).
(* alors, qu'importe le robogramme, l'exploration avec stop est impossible *)
```

Exprimer un contre exemple: la configuration Nous supposons que k divise n et nous définissons une configuration initiale où les robots sont tous équidistants les uns des autres, c'est-à-dire que chaque robot est à n/k du robot précédent, ainsi qu'un démon qui active tous les robots a chaque round.

Pour cela nous nous reposons sur l'arithmétique sur $\mathbb{Z}/n\mathbb{Z}$ pour affecter un sommet à chaque robot: la fonction `create_ref_config` prend en paramètre un robot (correct) `g`, qui est un élément de type `Fin.t k`. Ce type correspond à un entier naturel associé à une preuve, mais seul l'entier nous intéresse ici, et nous utiliserons donc la fonction `proj1_sig` pour l'extraire. Nous multiplions cet entier par n/k pour produire l'entier relatif caractérisant le sommet sur lequel le robot en question sera. Pour nous assurer que le résultat est bien sur l'anneau, nous utilisons la fonction `Ring.of_Z` qui remplace un entier relatif par un entier entre 0 et la taille de l'anneau.

La configuration initiale `ref_config` est alors obtenue en attribuant à chaque robot le sommet obtenu:

```
(* on prend un robot g, on utilise sa partie entiere *)
Definition create_ref_config (g : G) : location :=
Ring.of_Z (Z_of_nat (proj1_sig g * (n / kG))).

(* la configuration de depart dans nos contre-exemples *)
Definition ref_config : configuration :=
fun id ⇒ match id with
            | Good g ⇒ create_ref_config g
            | Byz b ⇒ dummy_val
          end.
```

Nous pouvons noter qu'il y a des positions associées aux robots byzantins hypothétiques (ici `dumy_val`, le sommet correspondant à l'entier 0). Cela est nécessaire car la fonction est indépendante de nos axiomes, notamment celui spécifiant qu'il n'y a aucun Byzantin. Ces valeurs seront non pertinentes et les cas concernant les byzantins dans la preuve seront toujours écartés directement sans être un fardeau.

Pour prouver que cette configuration est bien une configuration de départ valide, nous avons le lemme suivant:

```
Lemma ref_config_injective :
Util.Preliminary.injective eq equiv (fun id ⇒ get_location (ref_config id)).
```

Exprimer un contre exemple: le démon Nous définissons maintenant un démon qui, pris avec la configuration `ref_config` va définir le contre-exemple de l'exploration avec stop pour tout protocole. Notre but est de garantir que, si un robot bouge, alors tous les robots bougent dans la même direction, menant à une nouvelle configuration indistinguable de la configuration initiale `ref_config`. Ainsi, le démon devrait garantir que tous les robots:

1. Sont activés chaque round,
2. Perçoivent la même observation,
3. Bougent dans la même direction.

Pour que les robots aient tous la même perception, nous définissons pour chaque sommet `v` un isomorphisme `trans v` qui fait tourner le graphe en entier de telle façon que le sommet `v` soit envoyé sur l'`origin`. Le nom `trans` vient du fait que, dans l'espace $\mathbb{Z}/n\mathbb{Z}$ sous-jacent au graphe, cette "rotation" est implémenté comme une translation: la même valeur $v_{\mathbb{Z}}$ est soustraite à chaque sommet pour calculer son image.

Enfin, nous définissons comme suit une unique action démoniaque `da` qui sera utilisée à chaque round d'une exécution:

```
Program Definition da : demonic_action := {|
  activate := fun id ⇒ true;
  (* tous les robots sont actives *)
```

```
  relocate_byz := fun _ _ ⇒ dummy_val;
  (* une valeur fictive pour les byzantins *)
  change_frame := fun config g ⇒ (to_Z (config (Good g)), false);
  (* pour les bons robots, une valeur est donnee pour la translation *)
  choose_update := fun _ _ _ ⇒ tt;
  choose_inactive := fun _ _ ⇒ tt |}.
  (* il n'y a rien a faire pour les autres fonctions, car l'etat des
  robots ne contient que leur position dans cette preuve.*)
```

Ici `Program` est un mot clé de COQ permettant de laisser l'utilisateur prouver à la main les propriétés nécessaire à la définition.

Intéressons-nous à la partie concernant le changement de référentiel `change_frame` qui définit l'isomorphisme appliqué à la perception de ce robot: appliquée à un robot `g`, la fonction renvoie la translation à appliquer à l'observation du robot pour que tous voient la même chose.

Comme le démon `bad_demon : demon := Stream.constant da.` active tous les robots à chaque round, il est facile de prouver que c'est un démon FSYNC:

```
Lemma FYSNC_setting : FSYNC bad_demon.
```

La preuve formelle

Une propriété intéressante de notre configuration initiale `ref_config` et du référentiel donné par l'action démoniaque `da` aux robots est que tout robot dans `ref_config` aura exactement la même perception de son environnement que les autres (c'est-à-dire qu'il calculera sa destination en fonction de la même observation).

Pour tout robot `g` de la configuration `ref_config`, si sa perception de l'observation subit une translation le long de l'anneau telle que `g` se retrouve au niveau de l'origine, alors cette perception est équivalente à la perception sans translation.

```
Lemma obs_trans_ref_config : ∀ g,
!! (map_config (Ring.trans (to_Z (create_ref_config g))) ref_config) ≡ !! ref_config.
```

où `!! config` est une notation pour la fonction de calcul de l'observation de la configuration `config` au point `origin`:

```
Notation "!! config" := (obs_from_config config origin) (at level 0).
```

Nous déduisons de cette première propriété que, dans la configuration `ref_config` et avec l'action démoniaque `da`, tout robogramme `r` décide du même mouvement pour tous les robots.

Comme vu précédemment dans la partie 5.2, dans l'anneau les mouvements possibles sont limités à:

- Rester sur place en suivant l'arête `AutoLoop`, ce qui correspond dans cet anneau à ajouter à la valeur du nœud 0,
- Aller vers l'avant en suivant l'arête `Forward`, et ainsi changer la valeur en ajoutant 1 ou bien
- Aller vers l'arrière en suivant l'arête `Backward`, ce qui revient à ajouter (-1).

```
Lemma ring_range : ∀ (r : robogram) (r : robot),
  let m := r (spect_equi config_equi r) in
  m = 1_V ∨ m = 0_V ∨ m = (-1)_V.
```

Ces trois possibilités définissent une classification suffisante des protocoles, et il est possible de vérifier que dans chacun de ces trois cas, `r` ne résout pas l'exploration avec stop.

Soit `r` un robogramme, et m le mouvement commun calculé par `r` pour tous les robots.

Cas $m = 0$. Dans ce cas nous vérifions en premier lieu que l'exécution reste bloquée pour toujours dans la configuration `ref_config`.

```
Lemma round_id :
(* la configuration apres un round est equivalente [≡] a la
configuration [ref_config]  *)
  round r da ref_config ≡ ref_config.
```

Puis nous en déduisons qu'il existe au moins un sommet qui ne sera jamais visité:

```
Lemma NeverVisited_ref_config : ∀ e,
(* Pour toute execution, *)
e ≡ execute r bad_demon ref_config →
(* si cette execution commence a la configuration ref_config, et
utilise le robogramme r et le demon bad_demon, *)
∃ pt, ¬ Will_be_visited pt e.
(* alors il existe un point qui ne sera pas visite *)
```

Les sommets qui ne sont jamais visités sont précisément les sommets qui ne sont pas occupés dans la configuration initiale `ref_config`. Ceci est vrai par exemple pour le sommet 1_V.

Cas $m = 1$ (et $m = -1$). Dans ce cas chaque robot se déplace sur le sommet voisin sur sa droite. Cela a comme conséquence que la nouvelle configuration après un round est isomorphe à la configuration `ref_config`, par la fonction de translation `trans (-1)`$_V$.

Pour finir ce cas, nous prouvons que les configurations successives de l'exécution seront toutes équivalentes les unes aux autres pour que tous les robots continuent de bouger dans le même sens pour toujours. Ainsi, l'exécution ne s'arrêtera jamais dans ce cas.

Au centre de cette preuve réside la notion de configurations équivalentes contenant toutes les configurations qui seront présentes dans l'exécution. Nous définissons `equiv_config` c_1 c_2 comme vérifié quand la configuration c_2 peut être obtenue en appliquant, pour un certain `v`, l'isomorphisme `trans v` à c_1.

La propriété principale de cette relation d'équivalence est qu'avec l'action démoniaque `da`, un robot perçoit la même observation pour toute configuration équivalente.

```
Lemma config1_obs_equiv : ∀ config1 config2,
equiv_config config1 config2 →
∀ g, !! (map_config (trans (to_Z (config1 (Good g)))) config1)
       ≡ !! (map_config (trans (to_Z (config2 (Good g)))) config2).
```

En particulier, cela s'applique à `ref_config` et toute configuration `c` équivalente à `ref_config`: Nous avons donc exprimé le prédicat correspondant au fait qu'une exécution commençant par `ref_config` et utilisant le démon `bad_demon` traversera uniquement des configuration équivalentes à `ref_config`.

De là, nous en déduisons que tout robot continue de bouger, et que l'exécution ne s'arrête jamais. Cela prouve que `r` ne résout pas l'exploration avec stop, et donc s'applique à toute la classe de robogrammes définis par $m = 1$.

```
Definition AlwaysEquiv k (e1 e2 : execution) : Prop :=
  Stream.forever2 (Stream.instant2 (equiv_config_k k)) e1 e2.
(* forever2 est l'operateur de logique temporelle forever, mais
   dont la propriete utilise 2 executions.
   A tout moment dans les deux executions, les
   configurations a un meme moment sont equivalentes deux a deux par
   translation de ``k'' dans l'anneau.*)

Lemma AlwaysEquiv_ref_config :
  AlwaysEquiv (to_Z target) (execute r bad_demon ref_config)
```


Rapport-gratuit.com
LE NUMERO 1 MONDIAL DU MÉMOIRES

```
                              (Stream.tl (execute r bad_demon ref_config)).
(* Nous ne deduisons qu'en partant de la configuration ref_config,
   toute configuration dans la suite de l'execution sera equivalente a
   ref_config. *)

Definition AlwaysMoving (e : execution) : Prop :=
  Stream.forever (fun e1 ⇒ ¬Stopped e1) e.
(* Une execution est toujours en train de bouger si elle ne s'arrete jamais. *)

Lemma ref_config_AlwaysMoving : AlwaysMoving (execute r bad_demon ref_config).
(* Comme toute configuration est equivalente a ref_config et que
   dans ref_config, on bouge, alors l'execution ne s'arretera jamais. *)

Theorem no_exploration_moving : ¬ Explore_and_Stop r
(* Comme l'execution ne s'arrete pas, l'exploration avec stop n'est
   pas resolue. *)
```

Le cas $m = -1$ est similaire, avec la translation dans l'autre sens. Ceci conclu la preuve du théorème `no_exploration_k_divides_n`: aucun protocole ne peut résoudre l'exploration avec stop d'un anneau de taille n avec k robots si k divise n. Cette preuve a nécessité près de 200 lignes de code pour les définitions du problème et près de 600 lignes de code pour l'impossibilité si k divise n.

Chapter 6

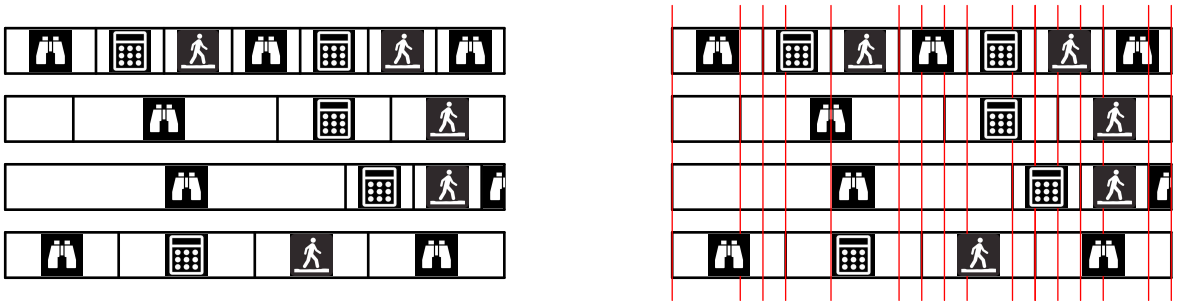
Exécutions ASYNC dans PACTOLE

Nous avons vu précédemment au chapitre 4 que le prototype original de PACTOLE contenait, avant nos travaux, deux modélisations pour la synchronisation des robots, le mode FSYNC et le mode SSYNC. Nous avons présenté le modèle ASYNC au paragraphe 2.3. Ce modèle est largement utilisé dans la littérature et afin de servir davantage de réalisme nous l'avons implémenté dans PACTOLE.

Nous décrivons cette intégration au modèle formel et illustrons son utilisabilité en établissant formellement une preuve d'équivalence entre deux modèles de graphes utilisant cette synchronisation.

6.1 Rappels sur ASYNC

Pour rappel, le modèle ASYNC a comme principe de ne plus utiliser la séparation du temps en round mais de rendre indépendants chaque action et chaque robot. Dans ce modèle, chaque robot réalise ses actions *Look*, *Compute* ou *Move* à son propre tempo, qui peut ne pas être celui des autres robots.



(a) Une exécution de quatre robots dans le modèle ASYNC.

(b) La même exécution, mais avec les lignes rouges marquant les différentes actions.

Figure 6.1

Dans la figure 6.1, les traits rouges représentent chaque instant où il y a un changement pertinent dans la configuration. Il est important de remarquer que, contrairement aux modes FSYNC et SSYNC, ce mode

Figure 6.2: Un exemple où pendant que le robot dans la ligne supérieure calcule sa prochaine destination, le robot de la ligne inférieure a bougé : le mouvement du premier robot est décidé en fonction d'informations périmées.

n'utilise plus la notion de *round*. De ce fait, il est possible voire même probable qu'un robot prenne une décision sur des informations périmées.

En effet, si un robot prend un snapshot de son environnement, et que pendant son temps de calcul, l'état du système est modifié, par exemple par un robot se déplaçant, alors le robot initial bougera vers une nouvelle position en fonction d'informations qui ne sont plus exactes au commencement de son mouvement (voir figure 6.2).

6.2 Implémentation

Les modèles asynchrones n'utilisent pas dans la littérature la notion de *round* mais il existe bien des changements dans l'exécution. Nous avons choisi de regarder l'état de l'exécution à chaque changement d'état d'un robot. Il faut re-vérifier l'état d'un robot dès que quelque chose change dans tout autre robot. Cela correspond à chaque trait rouge dans 6.1.

Le snapshot dans la phase *Look* est instantané. Il est donc possible de découper la phase *Look* en une période d'inactivité pour le robot suivie à la fin du temps défini pour l'action par un snapshot instantané comme présenté à la figure 6.4.

Il n'y a pas de différence observable entre un calcul qui s'étend entre deux instants t_1 et t_2 et un calcul instantané en t_1 suivi d'une période d'inactivité jusqu'à t_2. Il est donc possible de découper la phase *Compute* en un calcul instantané de la prochaine destination, suivi d'un temps d'inactivité du robot correspondant à son temps de calcul réel comme présenté à la figure 6.5.

Notre cycle est maintenant constitué de deux actions instantanées, des périodes d'inactivité et le mouvement. Ces périodes d'inactivité peuvent être considérées comme des moments où le robot est en mouvement mais où le mouvement est nul, sans perte de généralité. De plus, les deux actions instantanées étant juste l'une après l'autre, il est possible de les regrouper en une seule action *Look-Compute*.

Nous avons donc un modèle fonctionnant en alternance de deux états, un état où le robot est activé et fait l'action *Look-Compute*, et un état où le robot est inactif, et dans lequel il ne fait que se déplacer (6.3).

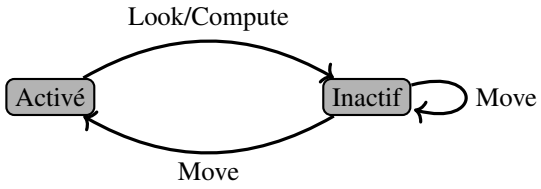


Figure 6.3: Les deux états du robot en ASYNC, et quelles actions mènent à quels états.

Pour implémenter le fait que l'action *Move* peut maintenant durer plusieurs activations du robot, nous ajoutons des informations dans le `state` des robots pour pouvoir continuer les mouvements en cours.

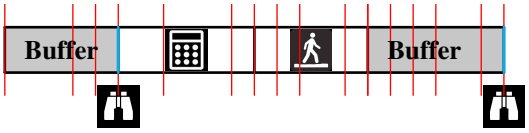


Figure 6.4: Le snapshot de la phase *Look* est ici instantané au niveau des traits |, et le temps associé au mouvement est remplacé par un buffer, c'est-à-dire une période d'inactivité du robot.

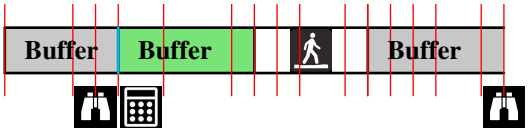


Figure 6.5: Le calcul de la phase *Compute* est instantané au même instant | que le snapshot ; le buffer, c'est-à-dire le temps d'inactivité du robot qui remplace le temps de calcul, est ici en vert.

Pour décrire le modèle ASYNC dans PACTOLE, nous rajoutons donc dans les états des robots le trajet à suivre tout au long du mouvement.

Dans le modèle ASYNC, la fonction de transition n'est plus entre deux rounds mais entre deux états du système comme présenté dans la figure 6.6. Il n'est donc plus possible de refaire l'observation et le calcul des nouvelles positions à chaque appel de la fonction `round` de PACTOLE car certains robots seront en mouvement.

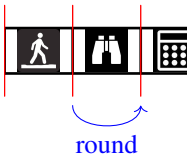


Figure 6.6: Un round dans le modèle ASYNC correspond à un changement dans l'état d'un des robots du modèle, et non plus un changement de cycle LCM comme précédemment. Ces rounds sont ici représentés par les traits rouges |.

L'idée de séparer les états du robots en deux nous permet d'utiliser ces deux états en les projetant sur nos deux états déjà existant dans le démon: l'état activé ou l'état inactif. Le choix de mettre un robot dans tel ou tel état est réalisé par la fonction `activate`.

Pour implémenter le fait que le mouvement s'étale sur une période où plusieurs observations par d'autres robots ont pu avoir lieu, nous choisissons de mettre les robots en mouvement comme des robots inactifs. Les robots actifs étant ceux qui font l'action *Look-Compute*. Comme la double action *Look-Compute* est considérée instantanée, il y aura aussi un premier mouvement lors de l'activation des robots. Ce premier mouvement peut sembler contredire qu'un robot peut se déplacer durant le calcul de la prochaine destination d'un autre robot mais ledit premier mouvement peut être nul si le démon représente le robot toujours en train de calculer.

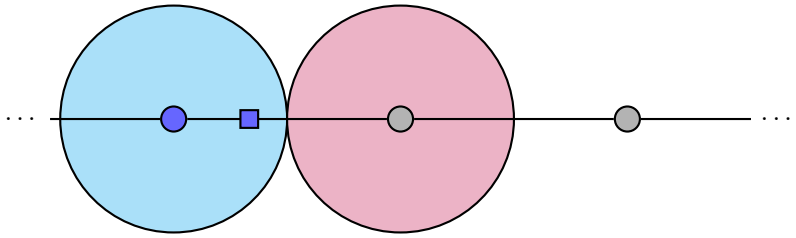


Figure 6.7: Représentation schématique du graphe continu à vision discrète: ■ représente le robot sur l'arête, ● représente un sommet du graphe et ◉ représente le sommet où est perçu le robot, avec la zone de perception. La limite de cette zone de perception est le seuil de l'arête, dès que le robot la dépasse et arrive dans la zone ● il n'est plus vu sur le même sommet.

6.3 ASYNC en action : une preuve d'équivalence

Pour illustrer ce fonctionnement de l'asynchrone, nous allons présenter une preuve d'équivalence entre deux modèles de graphes, le modèle de graphes discret et le modèle de graphe continu avec observation discrète (voir figure 6.7).

Dans les études du point de vue algorithmique distribuée des agents évoluant dans des graphes, les seuls déplacements considérés sont « plus » que rigides et on peut quasiment parler de téléportation : le *Move* est atomique d'un noeud à l'autre (mais peut survenir n'importe quand). On peut donc objecter que les aspects ASYNC sont fortement contraints car on ne peut pas voir un robot en train de se déplacer, comme on l'aurait en ASYNC dans un espace continu[1].

Nous proposons une preuve formelle d'équivalence établissant que les protocoles restent corrects si on considère des *Move* non atomiques mais rigides et observés de manière discrète et monotone.

Ce résultat s'appuie en particulier sur notre formalisation de ASYNC et sur notre modèles formel pour les graphes.

6.3.1 Modélisation des deux types de graphes

Il a été vu précédemment comment étaient construits les graphes dans le chapitre 5, avec les ensembles `V` et `E`. Pour nos deux modèles de graphes, nous avons donc deux types de positions différents: les positions sur un graphe discret 6.8a et les positions sur le graphe continu 6.8b.

Le type de `p` de l'arête est celui d'un ratio, un type représentant directement les nombres réels entre 0 et 1(6.9). Ce ratio représente quelle proportion de l'arête est déjà parcourue par le robot.

Notre modèle ne marche qu'avec des mouvement rigides, car il est nécessaire pour l'équivalence que les démons dans l'espace continu n'activent les robots que s'ils sont arrivés au bout de leur mouvement, c'est-à-dire sur un sommet (ce sont donc des déplacements rigides).

```
(* Si un robot est active...              alors il n'est pas sur une arete. *)
(∀ g, activate da (Good g) = true → ∀ e p, config (Good g) =/= SOnEdge e p).
```

Ces modèles devant pouvoir être modifiés par l'observation des robots, nous avons défini les bijections sur ces modèles de positions. Les bijections précédemment définies pour les graphes suffisent car nous ne modifions pas le ratio `p` déjà parcouru.

```
Definition bijectionG (iso : isomorphism G) : Bijection.bijection loc.
simple refine {| Bijection.section := fun pt ⇒ match pt with
          | OnVertex v ⇒ OnVertex (iso.(iso_V) v)
```

[1] On a cependant toujours la propriété que l'on peut baser son mouvement sur une observation obsolète.

```
(* un robot ne peut etre que
   sur un sommet *)
Instance LocationV : Location := {
  location := V }.

Notation locV :=
  (@location LocationV).
```

(a) Le type des positions dans un graphe discret.

```
Inductive loc :=
(* Un robot peut etre soit sur un sommet *)
  | OnVertex (l : location)
(* soit sur une arete. *)
  | OnEdge (e : E) (p : strict_ratio).

Instance LocationG : Location := {
  location := loc }.

Notation locG :=
  (@location LocationG).
```

(b) Le type des positions dans le graphe continu.

Figure 6.8: Les différents types de postions possibles suivant les graphes.

```
Definition ratio := {x : R | 0 ≤ x ≤ 1}ℝ.
Definition strict_ratio = {x : R | (0 < x < 1}ℝ.
```

Figure 6.9: Les définitions des ratios, par intention.

```
          | OnEdge e p ⇒ OnEdge (iso.(iso_E) e) p
        end;
Bijection.retraction := fun pt ⇒ match pt with
          | OnVertex v ⇒ OnVertex (Bijection.retraction iso.(iso_V) v)
          | OnEdge e p ⇒ OnEdge (Bijection.retraction iso.(iso_E) e) p
        end |}.
```

Dans les observations dans ces modèles, chaque robot est vu comme étant sur un sommet. Pour le modèle continu, ce sommet est défini en raison d'un seuil: un robot positionné à un certain ratio du chemin parcouru p le long de l'arête est perçu comme étant sur la source de l'arête si p est inférieur ou égal au `threshold` de l'arête, et sur la cible sinon. Le `threshold` est dans la définition du graphe comme un paramètre en plus prenant une arête et renvoyant un réel, ce réel étant assuré d'être dans l'intervalle $]0, 1[$ grâce à une propriété supplémentaire ajoutée à la classe d'un graphe.

```
threshold : E → R.
threshold_pos : ∀ e, (0 < threshold e < 1)ℝ.
```

L'observation dans le cadre du modèle discret ne sera pas explicitement définie, mais on calquera l'observation du modèle continu sur une instance de l'observation du modèle discret.

Dans nos modèles, les robogrammes renvoient une arête qui est l'arête à suivre pour le mouvement du robot. Cette arête est mise dans l'état du robot comme étant le chemin à suivre lors du mouvement.

Pour nous assurer que ces arêtes sont reliées à la position actuelle du robot, nous ajoutons l'arête à suivre dans les informations du robot, tout en ajoutant des propriétés de sûreté disant que si le robot est sur un sommet, l'arête à suivre est reliée à ce sommet.

Nous avons aussi ajouté des propriétés pour être sûr que les transformations données par le démon ne changent pas les valeurs de `threshold` et sont bien des isomorphismes.

```
(* Le sommet sur lequel est le robot est soit le point de depart de
   l'arete qu'il veut parcourir, soit sa destination. *)
Definition valid_stateV (state : locV * E) :=
```

```
  fst state ≡ src (snd state) ∨ fst state ≡ tgt (snd state).

Definition stateV := sig valid_stateV.

Local Instance InfoV : @State LocationV stateV.
(* Le mot cle [refine] permet d'enoncer une partie de la classe, et de
prouver par la suite les parties manquantes. Ici il restera a prouver
les propriete [lift_id], [get_location_lift], [get_location_compat] et
[lift_compat], necessaire a la bonne declaration d'un [State]. *)
simple refine {|
  (* La position dans l'espace des graphes discrets est donee par le
     sommet en premiere position dans la signature qu'est stateV. *)
  get_location := fun state ⇒ fst (proj1_sig state);
  state_Setoid := stateV_Setoid;
  (* Pour pouvoir relever les transformations de l'espace, ces dernieres
     doivent etre des isomorphismes comme defini dans le cadre des
     graphes, et ne doivent pas changer les threshold. *)
  precondition := fun f ⇒ sigT (fun iso ⇒ f ≡ iso.(iso_V)
                                        ∧ iso_T iso ≡ @Bijection.id R _);
  lift := fun f state ⇒ exist _ (projT1 f (fst (proj1_sig state)),
                    iso_E (projT1 (projT2 f)) (snd (proj1_sig state))) _ |}.


(* Dans le cas continu, l'etat doit aussi contenir la destination du robot.
   Si le robot est sur une arete, la destination est l'arrivee de cette arete.
   Si le robot est sur un sommet, on reprend comme pour le cas discret. *)
Inductive stateG :=
  | SOnVertex v e (proof : valid_stateV (v, e))
  | SOnEdge (e : E) (p : strict_ratio).

Definition good_iso_of f iso := f ≡ Bijection.section (bijectionG iso)
                                ∧ iso_T iso ≡ @Bijection.id R _.
Definition preconditionG := fun f ⇒ sigT (good_iso_of f).

Definition liftG (f : sigT preconditionG) state :=
  let iso := projT1 (projT2 f) in
  match state with
    | SOnVertex v e proof ⇒
        SOnVertex (iso_V iso v) (iso_E iso e) (valid_stateV_iso (v, e) iso proof)
    | SOnEdge e p ⇒ SOnEdge (iso_E iso e) p
  end.

  Definition stateG2loc state :=
  match state with
    | SOnVertex v _ _ ⇒ OnVertex v
    | SOnEdge e p ⇒ OnEdge e p
  end.

Local Instance InfoG : @State LocationG stateG.
simple refine {|
  (* La position est soit le sommet si on est sur un sommet,
     soit juste la position exacte sur l'arete.  *)
  get_location := stateG2loc;
```

```
  state_Setoid := stateG_Setoid;
  (* Les conditions pour relever les transformations dans ce modele
     sont les memes que pour le modele discret, un isomorphisme sur les
     elements du graphes, mais qui ne change pas les threshold. *)
  precondition := preconditionG;
  lift := liftG |}.
```

Il est maintenant possible de définir les types des configurations dans lesquels évolueront les robots dans chacun des modèles, et il est possible de traduire les `state` et les configurations d'un modèle à l'autre:

```
Notation configV := (@configuration _ stateV _ _).
Notation configG := (@configuration _ stateG _ _).

(* La traduction d'un etat discret vers un etat continu revient a integrer
   l'etat discret dans le constructeur pour sommet de l'etat continu. *)
Definition state_V2G (state : stateV) : stateG :=
  SOnVertex (fst (proj1_sig state)) (snd (proj1_sig state)) (proj2_sig state).

(* La traduction de l'etat continu vers l'etat discret est la projection
   du sommet ou de la position sur l'arete vers le sommet correspondant. *)
Definition state_G2V (state : stateG) : stateV :=
  match state with
    | SOnVertex v e p ⇒ exist valid_stateV (v, e) p
    | SOnEdge e p ⇒ if Rle_dec (@threshold locV E G e) p
                    then exist valid_stateV (Graph.tgt e, e) ltac:(now right)
                    else exist valid_stateV (Graph.src e, e) ltac:(now left)
  end.

Definition config_V2G (config : configV) : configG :=
  fun id ⇒ state_V2G (config id).

Definition config_G2V (config : configG) : configV :=
  fun id ⇒ state_G2V (config id).
```

Figure 6.10: Définitions et fonctions de traduction pour les états et les configurations dans les deux modèles.

L'observation dans le modèle continu est pratiquement la même que dans le modèle discret, avec si on est sur une arête, la projection soit sur le point de départ, soit sur l'arrivée de l'arête, en fonction du `threshold`.

```
Global Instance obs_V2G (Spect : @Observation _ _ InfoV _)
  : @Observation _ _ InfoG _.
simple refine {|
  observation := @observation _ _ _ _ Spect;
  obs_from_config := fun config st ⇒
    obs_from_config (config_G2V config) (state_G2V st);
  obs_is_ok s config st := obs_is_ok s (config_G2V config) (state_G2V st) |}.

Context {Obs : @Observation _ _ InfoV _}.
Notation robogramV := (@robogram _ _ InfoV _ Obs E _).
Notation robogramG := (@robogram _ _ InfoG _ (obs_V2G Obs) E _).
```

Les robogrammes recevront ainsi les mêmes informations avant de prendre une décision, et renverront un choix de type `E`, qu'importe le modèle.

Lorsqu'un robot est en mouvement, les fonctions `choose_update` et `choose_inactive` renvoient les informations nécessaires à ce mouvement. Dans le cadre du modèle discret, ces fonctions renvoient un booléen, représentant le mouvement vers la destination si le booléen est à `true`. Dans le cadre du modèle continu, les fonctions renvoient un ratio m correspondant au pourcentage de l'arête parcouru, selon le schéma suivant:

- Si le robot est sur le sommet source de son mouvement à venir, $m = 0$ signifie rester sur place, $m = 1$ signifie se rendre directement sur le sommet destination, et $0 < m < 1$ signifie aller à la position correspondante le long de l'arête entre le sommet source et le sommet destination.
- Si le robot est sur une position p quelconque sur une arête, il ira sur la position $m + p$ sur cette même arête. Si $m + p >= 1$, alors le robot se retrouve sur le sommet destination.
- Si le robot est déjà sur le sommet destination, alors il reste sur place.

6.3.2 Traductions entre les modèles

Pour démontrer que le modèle discret et le modèle continu à vision discrète sont équivalents pour les robots oublieux, nous prouvons que tout robogramme produit la même exécution dans les deux modèles.

On remarquera dans un premier temps qu'un robogramme dans l'un des modèles peut aussi être traduit dans un robogramme de l'autre modèle grâce aux faits suivants:

- Le premier paramètre d'un robogramme est l'observation, et le type de l'observation est le même dans les deux modèles, grâce à la vision discrète de notre modèle continu.
- La position actuelle du robot est toujours un sommet puisque le modèle général suppose que le robogramme est appliqué seulement aux robots actifs qui, eux, ne sont positionnés que sur des sommets grâce aux propriétés du démon.
- La destination renvoyée par le robogramme est une arête.

```
Definition rbg_V2G (rbgV : robogramV) : robogramG :=
  @Build_robogram _ _ InfoG _ (obs_V2G Obs) _ _ rbgV rbgV.(pgm_compat).

Definition rbg_G2V (rbgG : robogramG) : robogramV :=
  @Build_robogram _ _ InfoV _ Obs _ _ rbgG rbgG.(pgm_compat).
```

Les traductions pour les configurations et les états ont été vus précédemment dans le code de la figure 6.10.

La traduction des démons se fait en prenant un démon d'un des modèles, et en traduisant chacune des fonctions caractérisant le démon, et en prouvant par la suite que ces fonctions respectent les propriétés nécessaires comme vu dans la figure 6.11.

6.3.3 Équivalence des modèles

Il reste à prouver l'équivalence des modèles. Pour cela, nous allons prouver la propriété qui se lit maintenant comme suit: pour tout robogramme `rbg`, l'action démoniaque `da` et la configuration `c` dans le modèle discret, il existe une action démoniaque `da'` dans le modèle continu telle que le diagramme de la figure 6.12 est satisfait.

Pour la preuve d'équivalence, nous prouvons chacun des sens séparément. Commençons par le sens discret vers continu avec vision discrète. Nous avons donc un théorème `graph_equivD2C` comme suit: pour toute configuration `config`, robogramme `rbg` et action démoniaque `da` dans le modèle discret, la traduction du résultat de la fonction de transition `round` utilisant `config rbg` et `da` équivaut au résultat de la fonction de transition utilisant les traductions de `rbg, da` et `config`

```
Theorem graph_equivD2C : ∀ (config : DGF_config) (rbg : robogramV) (da : DGF_da),
  config_V2G (round rbg da config) ≡ round (rbg_V2G rbg) (da_D2C da) (config_V2G config).
```

La preuve se résume à un raisonnement par cas sur les différents paramètres de la fonction de transition: le robot est-il activé? si oui, est-il byzantin ou non? le ratio du mouvement est-il 1 ou 0? De là, nous en déduisons que toute exécution dans le modèle discret peut être simulée dans le modèle continu.

La propriété réciproque est plus complexe. Il est nécessaire de rajouter en premier lieu la propriété spécifiant qu'un robot ne peut être activé qu'une fois sur un sommet du graphe. Deuxièmement, de par notre implémentations du démon continu, il nous faut rajouter une hypothèse: les décisions du démon ne prennent en compte que les informations de positions accessibles par les robot. Cela signifie que les décisions de ce démon ne peuvent prendre en compte que les positions vues dans le modèle discret:

```
Theorem graph_equiv_C2D : ∀ (config : CGF_config) (rbg : robogramG) (da : CGF_da),
(** les choix du demon dependent seulement des positions discretes *)
(∀ g, change_frame da (config_V2G (config_G2V config)) g  ≡ change_frame da config g) →
(∀ config g x, (* Il est necessaire de quantifier sur [config]
                        car c'est la configuration locale qui est utilisee *)
   choose_update da (config_V2G (config_G2V config)) g x ≡ choose_update da config g x) →
(∀ b, relocate_byz da (config_V2G (config_G2V config)) b ≡ relocate_byz da config b) →
(∀ id, choose_inactive da (config_V2G (config_G2V config)) id
                ≡ choose_inactive da config id) →
(** les robots sont uniquement actives sur les sommets *)
(∀ g, activate da (Good g) = true → ∀ e p, config (Good g) =/= SOnEdge e p) →
config_G2V (round rbg da config)
           ≡ round (rbg_G2V rbg) (da_C2D da config) (config_G2V config).
```

Ici encore, la preuve du passage du modèle continu au modèle discret est une preuve par cas sur tous les paramètres de la fonction de transition, qui sont encore plus nombreux que dans le cas précédent, la définition de l'action démoniaque `da` distinguant beaucoup plus de cas que précédemment.

Ces deux résultats, pris ensemble, disent que toute exécution, qu'importe le modèle (discret ou continu) peut être relié à une exécution équivalente dans l'autre modèle.

Nous avons donc établi formellement un premier pont entre deux modèles précédemment distincts pour les robots oublieux. Cette preuve, sans prendre en compte la partie formalisation des modèles, a pris 200 ligne de codes pour la partie discrète vers continue, et 400 dans le cas continu vers discret, en grande partie dues au plus grand nombres de possibilités dans le modèle continu, ce qui augmente le nombre de cas à étudier. D'un point de vu pratique, l'équivalence formelle que l'on fournit entre ces modèles ouvre

de nouveaux horizons sur ce qui est réellement calculable dans un environnement réel par des robots aux capacités limités.

```
Notation DGF_da :=
  (@demonic_action _ _ InfoV _ _ _ _ _ _ FrameChoiceIsomorphismV
      graph_update_bool graph_inactive_bool).
Notation CGF_da :=
  (@demonic_action _ _ InfoG _ _ _ _ _ _ FrameChoiceIsomorphismG
      graph_update_ratio graph_inactive_ratio).

(* traduction du modele discret vers le modele continu *)
Definition da_D2C (da : DGF_da) : CGF_da.
simple refine {|
  (* les memes robots sont actives dans chacun des modeles *)
  activate := da.(activate);
  (* traduction de l'emplacement renvoye pour les byzantins *)
  relocate_byz := relocate_byz_D2C da.(relocate_byz);
  (* le changement de referentiel se fait en fonction de la traduction
     de la configuration *)
  change_frame := fun config g ⇒ da.(change_frame) (config_G2V config) g;
  (* le demon du modele discret renvoie un booleen pour savoir si le
     robot bouge ou non, c'est traduit dans le modele continu par un
     ratio de 1 ou 0. *)
  choose_update := fun config g target ⇒
    if da.(choose_update) (config_G2V config) g target
    then ratio₁ else ratio_0;
  (* idem pour les robots inactifs *)
  choose_inactive := fun config id ⇒
    if da.(choose_inactive) (config_G2V config) id
    then ratio₁ else ratio_0 |}.

(* Ici, la configuration est necessaire car il nous faut savoir si un
   robot est sur une arete pour [choose_inactive] *)
Definition da_C2D (da : CGF_da) (Cconfig : CGF_config) : DGF_da.
simple refine {|
  (* les meme robots sont actives *)
  activate := da.(activate);
  (* traduction de l'emplacement renvoye pour les byzantins*)
  relocate_byz := relocate_byz_C2D da.(relocate_byz);
  (* le changement de referentiel se fait en fonction de la traduction
     de la configuration*)
  change_frame := fun config g ⇒ da.(change_frame) (config_V2G config) g;
  (* le demon discret renvoi le booleen traduisant si le robot, apres
     le mouvement decide par le demon continue, est avant ou apres le [threshold]. *)
  choose_update := fun config g target ⇒
    Rle_bool (threshold target) (da.(choose_update) (config_V2G config) g target);
  (* pour les robots inactifs, *)
  choose_inactive := fun config id ⇒
  (* nous commencons a chercher ou se situe le robot le long de l'arete, *)
    let current_ratio :=
      match Cconfig id with (* la configuration continue est utilisee ici *)
        | SOnVertex v e proof ⇒ ratio_0
        | SOnEdge e p ⇒ p
      end in
      let e := snd (proj1_sig (config id)) in
  (* puis on calcul la nouvelle position apres la modification par le
     demon continu, pour enfin renvoyer le booleen traduisant si cette
     nouvelle position est avant ou apres le [threshold] *)
    Rle_bool (threshold e)
             (current_ratio + da.(choose_inactive) (config_V2G config) id) |};
```

Figure 6.11: Fonctions de traduction pour les actions démoniaques.

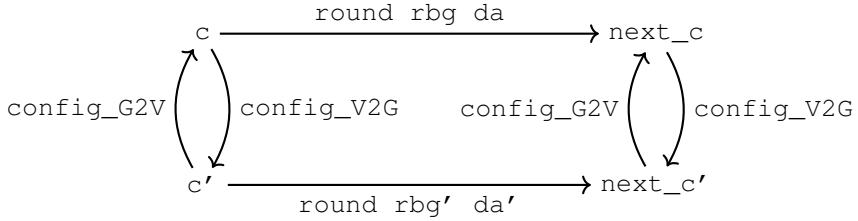


Figure 6.12: Bisimulation: pour tout robogramme `rbg`, l'action démoniaque `da` et la configuration `c` dans le modèle discret, la traduction de la configuration `next_c` résultant de la fonction de transition `round` équivaut au résultat `next_c'` de la fonction de transition appliquée aux traductions de chacun des trois paramètres `da' rbg'` et `c'` dans le modèle continu. L'inverse est vrai aussi, en traduisant dans l'autre sens chacun des paramètres.

Chapter 7

Étude de cas : maintien de connexion dans un réseau dynamique

Nous illustrons certaines des possibilités offertes par le cadre formel en étudiant un scénario d'application des essaims de robots : le maintien d'une connexion entre une cible aux déplacements arbitraires (dans des limites de vitesse acceptables) et une base fixe, à l'aide d'une chaîne de robots autonomes.

Ce cas d'application peut par exemple servir à suivre une équipe de recherche et secours tout en lui apportant un réseau ad-hoc de communication avec le centre de ressources.

Notre étude est découpée en deux parties : la poursuite d'une cible par un groupe de robots, sans collision, puis son enrichissement avec l'émission possible de nouveaux robots (volumiques) par une base fixe.

Dans notre problème, nous voulons que l'essaim suive une cible présentant les mêmes caractéristiques de mouvement que les robots, en particulier la vitesse de déplacement. Nous la désignerons en conséquence comme *robot libre*, non soumis au protocole. Nous choisissons également cette dénomination pour éviter les confusions avec les objectifs de déplacements (donc *cibles*) des robots qui ne perçoivent pas le robot libre.

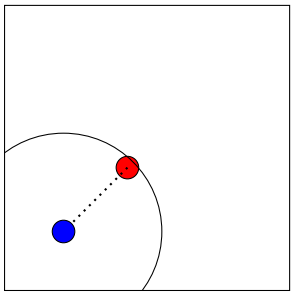


Figure 7.1: Le robot libre ● est perçu par le robot ● car il est dans sa zone de détection ○.

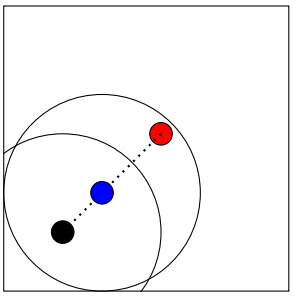


Figure 7.2: Nous cherchons à toujours avoir tous les robots dans le graphe de visibilité d'un robot percevant le robot libre. Le robot ● est ainsi lié au robot libre car dans le graphe de visibilité de ●.

Hypothèses

- L'**espace** considéré est $\mathbb{R}^2$.
- Les robots sont **oublieux**. Ils ne sont pas opaques. Nous leur ajouterons effectivement un volume dans la deuxième partie.
- Ils sont **identifiés** par un numéro strictement positif et équipés d'une **lumière** à deux états distinguables.
- Le robot libre est perçu comme un robot d'identifiant 0.
- Les capteurs ont tous la même **portée limitée** ; ils détectent identifiants et couleurs. Les figures 7.1 et 7.2 montrent des exemples de vision des robots.
- Les robots ont tous la même vitesse maximale de déplacement.
- Le mode de **synchronisation** est FSYNC[1].
- Les robots disposent d'un **dispositif d'élimination** arbitraire qui leur permet de se retirer de l'espace considéré. On peut dans le cas d'une application réelle penser à un changement de plan d'altitude, etc.

7.1 Protocoles de poursuite et connexion

7.1.1 Définitions

Les robots pouvant être sujets à des collisions, nous définissons les zones dans lesquelles ils sont respectivement en quasi-contact et en danger. Nous introduisons les zones de poursuite de cible et de relais.

- Nous nommons D la distance maximale que tout robot peut parcourir en un round.
- $Dmax$ correspond au rayon de visibilité défini au chapitre 2. Cette distance dépend ici de D, car pour éviter les collisions mais toujours garder en vue d'autres robots, il est nécessaire d'avoir une vision $Dmax \geq 6 \times D$. Cette valeur de $Dmax$ nécessaire pourra évoluer suivant les problèmes.

À partir de ces distances, nous définissons quatre zones:

Définition 7.1.1 (Zone de *Mort*.) *La zone de mort est le disque de rayon D autour du robot. Si un autre robot se trouve dans la zone de mort d'un robot donné, ces deux robots peuvent se percuter et ainsi faire échouer la tâche.*

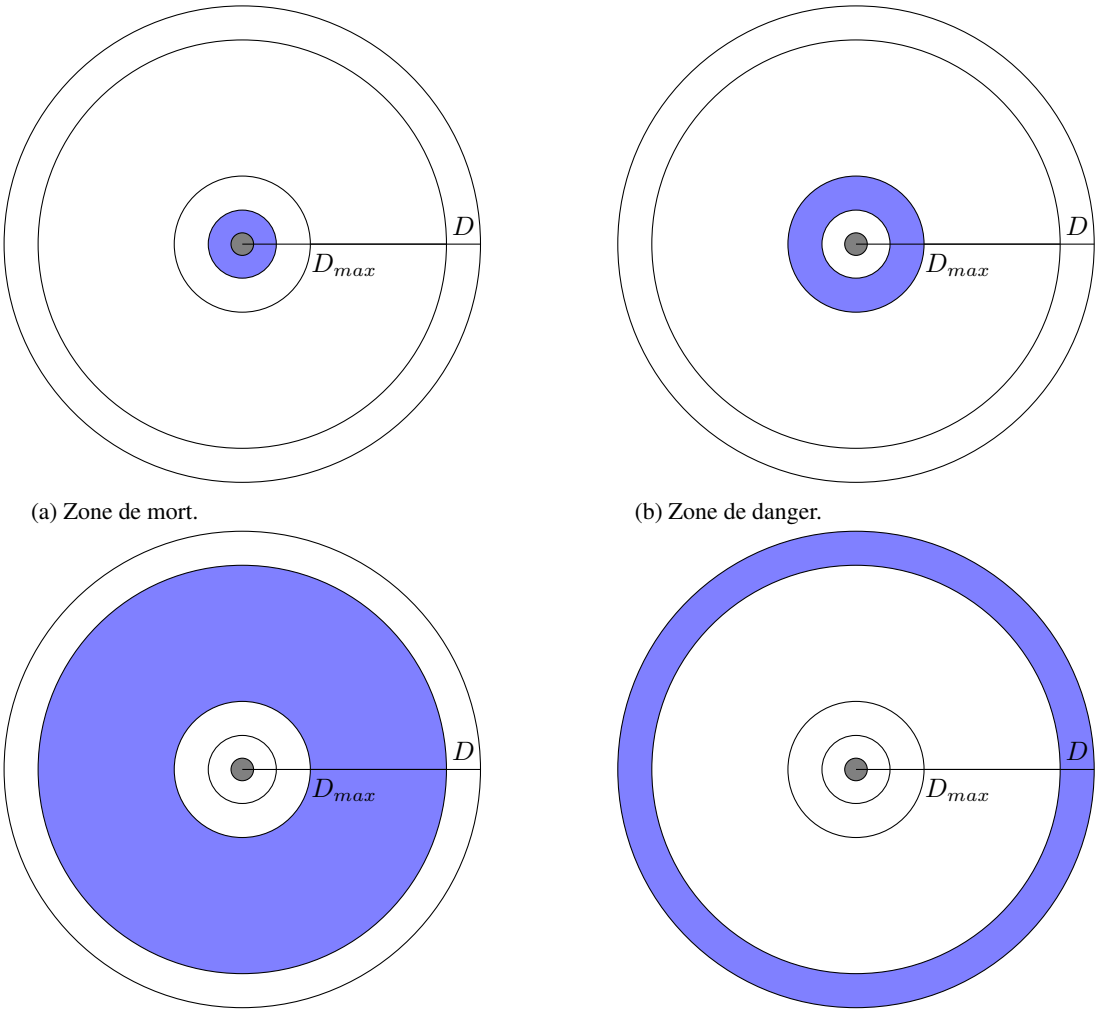
Définition 7.1.2 (Zone de *Danger*.) *La zone de danger est la zone entre les cercles de rayon D et $2 \times D$. Si un robot est dans cette zone, il faut y faire attention car il risque de rentrer dans la zone de* Mort.

Définition 7.1.3 (Zone de *Poursuite*.) *La zone de poursuite est la zone entre les cercles de rayon $Dmax$ et $Dmax - D$. Si un robot est dans la zone de poursuite d'un robot t et ne doit pas être perdu de vue, il faut que t bouge vers lui car sinon il risque de sortir de son champ de vision. Nous noterons $Dp = Dmax - D$ la distance de poursuite.*

[1] Un mode ASYNC rendrait la vitesse maximale des robots dépendante de leur taux d'activation. Ce serait le cas également pour un mode SSYNC sans hypothèse supplémentaire de type k-équitable, auquel cas la borne k laisse supposer une extension de notre résultat (au prix cependant de fastidieuses manipulations arithmétiques).

Définition 7.1.4 (Zone de *Relais*.) *La zone de relais correspond à la surface entre les cercles de rayons* $2 \times D$ *et* Dp*. Les robots dans cette zone servent de relais : ce sont des nœuds dans le graphe de visibilité.*

Ces zones sont schématisées dans la figure 7.3.



(a) Zone de mort.

(b) Zone de danger.

(c) Zone de relais.

(d) Zone de poursuite.

Figure 7.3: Les différentes zones du robot (en bleu).

7.1.2 Le choix de la cible

Le problème majeur rencontré dans cette tâche consiste à éviter les collisions. Il n'est pas raisonnable de prendre en compte les mouvements possibles de tous les robots visibles. Il serait en effet très facile d'arriver dans des situations de blocage comme présenté à la figure 7.4.

Nous choisissons de ne prendre en compte, dans le calcul de destination d'un robot r, que les robots visibles d'identifiant $r' < r$. Le robot libre est en particulier toujours pris en compte lorsqu'il est visible.

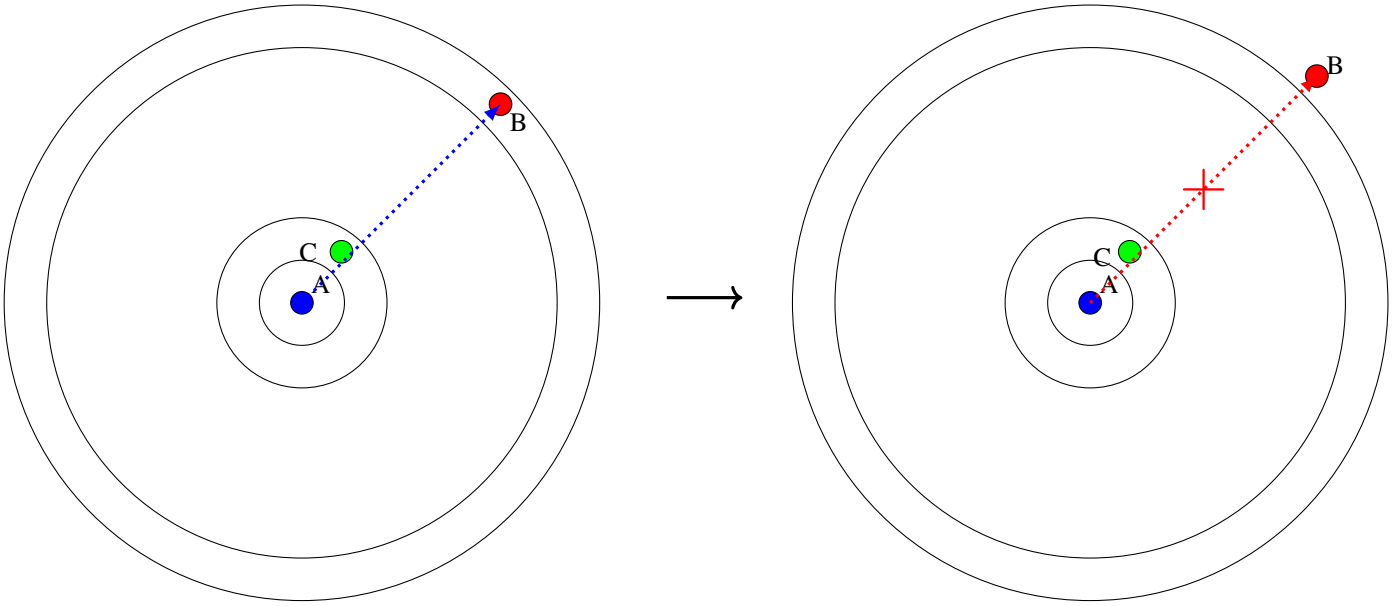


Figure 7.4: si le robot A veut se rapprocher du seul robot d'identifiant plus petit visible B mais que, sur son chemin, un robot d'identifiant supérieur C menace de rentrer dans la zone de mort, notre premier robot ne peut pas bouger et risque de perdre de vue le robot d'identifiant inférieur donc de casser le chemin entre lui et le robot libre.

Les robots d'identifiants inférieurs à r ne prennent pas r en compte pour leurs déplacements, c'est donc à r de sortir de leur chemin éventuel.

Pour obtenir un résultat le plus général possible, c'est-à-dire sans fournir de code de protocole mais plutôt des propriétés à respecter, nous nous interdisons de deviner comment les robots vont bouger. Dès lors, il est indispensable pour chaque robot de bouger à une position pour laquelle aucun robot d'identifiant inférieur ne peut se trouver dans la zone de *Mort*.

Ne connaissant pas le comportement des robots d'identifiants inférieurs, il est possible qu'ils se rapprochent de la position choisie, et donc il faut rajouter la zone de *Danger* à celle de *Mort* dans les zones à éviter. Il faut également toujours maintenir la connexion avec la cible choisie au départ, et donc rester à une distance maximale de Dp de la position actuelle de notre cible pour forcément la garder en vue au tour suivant.

Que se passe-t-il s'il n'existe aucune position répondant à ces critères?

Il n'est pas possible de supprimer du réseau tout robot ne trouvant pas de position adéquate pour éviter tout risque de collision. Une suppression si directe risque de casser la chaîne.

Une solution est apportée ici par les lumières: un robot ne pouvant pas bouger à une position sûre allume une lumière pour indiquer qu'il est en danger et ne bouge pas. Si lors du calcul des destinations au tour suivant, un robot se trouve dans sa zone de *Mort*, alors il se désactive directement et donc nous ne prenons pas le risque d'avoir des collisions.

Avec l'ajout de cette nouvelle donnée, nous affinons le choix de la cible: la cible choisie par un robot est prioritairement éteinte, c'est-à-dire qu'elle n'était pas en danger au tour précédent. Si toutefois il n'existe que des robots allumés dans le champ de vision d'un robot, ce dernier privilégiera comme cible un robot à moins de Dp de lui.

Nous montrerons lors de la preuve du lemme `exists_at_less_round` de la section suivante 7.2 que, même s'il n'existe dans le champ de vision que des robots allumés, il en existe au moins un à moins

de Dp.

Un autre cas extrême est à prendre en compte: que ce passe-t-il si le robot ne voit aucun autre robot ? Dans ce cas là, le robot doit s'éliminer car il ne peut plus aider à la poursuite du robot libre. Ce cas se révélera toutefois impossible à atteindre en partant d'une configuration correcte.

Pour résumer nous prétendons que tout protocole respectant les étapes suivantes remplit notre objectif :

1. Le robot fait un snapshot de son environnement en ne prenant que les robots d'identifiants inférieurs.
2. Pendant la phase *Compute*, s'il trouve un robot à moins de D ou s'il ne trouve aucun autre robots, il s'élimine[2].
3. Le robot choisit le robot qu'il veut suivre, forcément un robot d'identifiant inférieur.
4. Ayant choisi le robot à suivre pour cible, le robot détermine maintenant une position à moins de Dp de sa cible, et à moins de D de sa position actuelle.
   (a) S'il existe un robot d'identifiant inférieur à moins de $2 \times D$ de cette position choisie, alors le robot ne bouge pas et allume sa lumière,
   (b) Sinon le robot bouge vers cette position.

Notre objectif étant d'avoir à chaque point de l'exécution un ensemble de robots pour lequel le robot libre est dans le graphe de visibilité et chaque robot de cet ensemble se déplace en évitant les collisions.

7.2 Preuve formelle

Nous voulons prouver formellement qu'il existe toujours un chemin dans le graphe de visibilité entre un robot et le robot libre, mais aussi qu'à aucun moment, deux robots ne rentrent en collision. Pour ce faire, nous utilisons PACTOLE.

7.2.1 Définitions COQ

Nous commençons avec un nombre `n` non défini de robots, parmi lesquels il n'existe aucun byzantin:

```
Parameter n: nat.

(* Il y a n "bons" robots et 0 byzantin *)
Instance Identifiers : Names := Robots n 0.
```

L'état du système dans PACTOLE

Puis, nous définissons l'espace dans lequel se déplacent les robots comme suit:

```
Instance Loc : Location := make_Location (R2).
```

Comme nous l'avons vu en posant nos hypothèses, les robots n'ont pas accès uniquement à leur position mais aussi à d'autres informations: leur lumière, que nous caractériserons par un booléen, valant `true` si cette lumière est allumée, ainsi que l'identifiant de ce robot. Nous ajoutons enfin un deuxième booléen valant `true` pour un robot considéré comme *vivant*, ou `false` pour un robot éliminé.

[2] Il utilise le dispositif arbitraire de suppression supposé.

```
Definition identifiants := nat.
(*variable pour savoir si les robots sont allumes ou pas:
Si la variable de type [light] est a true, il est allume*)
Definition light := bool.
Definition alive := bool.
```

Une configuration est la position dans $\mathbb{R} \times \mathbb{R}$ avec en plus les informations concernant l'identifiant, la lumière et le statut vivant ou non du robot.

Nous avons donc d'une part la position définie dans `R2` (notation Coq de $\mathbb{R} \times \mathbb{R}$) ainsi que d'autre part le reste des informations nécessaires au bon déroulement du protocole dans un type nommé `ILA` (pour Identifiant Light Alive). Ce type est un triplet comprenant un entier naturel représentant l'identifiant du robot et deux booléens représentants respectivement si la lumière du robot est allumée et si le robot est vivant.

Nous avons donc l'état du système suivant:

```
(* Notre etat a comme type des positions R2, et comme type
d'informations accessibles (R2*ILA) *)
Instance State_ILA : @State (make_Location R2) (R2*ILA) :=
(* il est cree en creant en premier lieu un etat ou seul les
positions (R2) sont decrites, puis en ajoutant les informations
(ILA) *)
@AddInfo Loc R2 ILA _ _ (OnlyLocation (fun _ ⇒ location)).
```

Nous avons aussi les fonctions permettant d'accéder à chacune de ces informations: `get_ident`, `get_light` et `get_alive`. À noter qu'il existe aussi une fonction `get_location` permettant d'accéder à la position du robot déjà implémentée dans la notion du `State`.

Le robogram

Le robogram renvoie une position à atteindre pour le prochain round ainsi que l'état de la lumière du robot.

Le robogram commence par choisir une cible grâce à la fonction `choose_target`. Cette dernière n'est pas précisément définie car nous voulons être généralistes : elle est juste spécifiée par des axiomes. Ces axiomes sont définis à la figure 7.5.

Avec le robot pris pour cible déterminé, le robogram choisit une position vers laquelle aller, grâce à `choose_new_pos`, pour après décider si la position choisie est une position valable grâce à `move_to`, avec là encore de la liberté d'adaptation avec juste des spécifications données à la figure 7.6.

Nous pouvons donc définir le programme du robogram comme suit:

```
Definition rbg_fnc (s:obs_ILA) : R2*light :=
    (* Le robogram choisit la cible *)
    let target := choose_target s in
    (* Il decide de la position ou le robot veut etre au prochain round  *)
    let new_pos := choose_new_pos s (fst target) in
    match move_to s new_pos with
    (* Si le robot a pu bouger, il change la position,
       et n'allume pas la lumiere du robot. *)
    | true ⇒ (new_pos,false)
    (* si le robot n'a pas pu bouger, il l'allume. *)
    | false ⇒ ((0,0)ℝ, true)
    end.

Definition rbg_ila : robogram :=
  {| pgm := rbg_fnc |}.
```

La fonction `update`

Pour décider ce qui se passe après le choix de la prochaine position, une fonction « met à jour » la configuration.

Dans cette fonction, nous testons si un robot doit s'eliminer et nous mettons ses informations à jour suivant le résultat:

```
Definition upt_aux (config:config) (g: G)
   (rl : R2*light)(* le resultat du robogram *): (R2*ILA) :=
  match config (Good g) with (r,((i,l),a)) ⇒
(* cette fonction teste si un robot est trop pres, ou si aucun robot
     n'est a portee *)
  if upt_robot_dies_b config g
(* le robot meurt et donc ne bouge pas *)
  then (r,((i,l),false))
(* nous changeons les parametres du robot selon ce que renvoie le robogram *)
  else (fst rl,((i,snd rl), a))
  end.
```

Les propriétés du démon

Le démon utilisé dans cette preuve possède également des contraintes à respecter notamment pour garantir que le changement de référentiel est valide au sens où il place toujours le robot auquel est appliqué le changement à l'origine.

Les propriétés du démon sont regroupées dans le prédicat nommé `da_predicat` et nous définissons donc le type des démons utilisés dans cette preuve comme suit:

```
Definition demon_ILA (demon : demon_ila_type) :=
       Stream.forever (Stream.instant da_predicat) demon.
```

7.2.2 Prédicats

Nous voulons prouver que tout au long de l'exécution, aucun robot ne rentre en collision avec un autre, et que le robot libre est toujours dans le graphe de visibilité de tous les robots.

```
  Definition NoCollAndPath e :=
(* Tout au long de l'execution *)
  Stream.forever
(* il est vrai a l'instant observe *)
    (Stream.instant
(* qu'aucun robot ne rentre en collision *)
       (fun conf ⇒ no_collision_conf conf
(* et qu'il existe un robot d'identifiant inferieur visible *)
                   ∧ path_conf conf) ) e.
```

Il faut donc qu'en commençant dans une configuration initiale valide, avec un démon respectant nos contraintes, et en utilisant notre protocole, que l'exécution vérifie la propriété `NoCollAndPath`.

```
Theorem validity_conf_init:
  ∀ demon,
  demon_ILA demon →
  NoCollAndPath (execute rbg_ila demon config_init).
```

Avec ces définitions dans Coq, il est maintenant possible de définir les prédicats présentés précédemment:

```
(* choose_target prend l'observation et renvoie les informations d'un
robot a prendre comme cible  *)
Context {choose_target : obs_ILA → (R2*ILA)}.

(* Compatibilite de choose_target *)
Context {choose_target_compat : Proper (equiv ==> equiv) choose_target}.

(* On ne prend que des robots vivants *)
Axiom choose_target_alive : ∀ obs target,
    choose_target obs ≡ target
   → get_alive target ≡ true.

(* Si un robot pris pour cible est allume, tous les autres robots
visibles sont allumes *)
Axiom light_off_first: ∀ obs target,
    choose_target obs ≡ target →
                      get_light (target) ≡ true →
                      (* il n'y a pas de robots etein *)
                      For_all (fun (elt: R2*ILA) ⇒
                      get_light elt ≡ true) obs.

(* Le robot pris pour cible est toujours un robot d'identifiant inferieur. *)
Axiom target_lower : ∀ obs target (zero: R2*ILA),
        In zero obs → get_location zero ≡ (0,0)ℝ →
        choose_target obs ≡ target →
        get_ident target < get_ident zero.


(* Si le robot pris pour cible est allume et a plus de Dp,
 il n'existe pas de robot a moins de Dp *)
Axiom light_close_first : ∀ obs target,
    choose_target obs ≡ target →
    get_light target ≡ true →
    ((dist (0,0)ℝ (get_location target)) > Dp)ℝ →
    For_all (fun (elt : R2*ILA) ⇒
               ((dist (0,0)ℝ (get_location elt)) > Dp)ℝ) obs.

(* Le robot pris pour cible est forcement visible *)
Axiom choose_target_in_obs : ∀ obs target,
    choose_target obs = target  →
    In target obs.
```

Figure 7.5: Les axiomes et propriétés pour la fonction `choose_target`.

```
Context {move_to: obs_ILA → location → bool }.

Context {move_to_compat : Proper (equiv ==> equiv ==> equiv) move_to}.

(* Si le robot bouge, il n'existe pas de robot a moins de 2*D *)
Axiom move_to_Some_zone : ∀ obs choice,
    move_to obs choice = true →
    (∀ x, In x obs → let (pos_x, light_x) := x in
                          dist pos_x (0,0) > 0 →
                          dist choice pos_x > 2*D)%R.

(* Si le robot en peut pas bouger, alors : *)
Axiom move_to_None :
  ∀ (obs : obs_ILA) (choice : location),
  move_to obs choice = false →
(* il existe un autre robot *)
∃ other : R2 * ILA,
(* dans l'observation *)
(In other obs)%set ∧
(* tel qu'il ne soit pas le robot avec l'identifiant le plus haut
et ainsi il n'est pas le robot faisant l'observation *)
(∃ inf, In inf obs ∧ get_ident other < get_ident inf) ∧
(* et ce robot est a moins de 2*D de la position d'arrivee choisie *)
(dist (get_location other) choice ≤ 2*D)%R.

Context {choose_new_pos: obs_ILA → location → location}.
Context {choose_new_pos_compat : Proper (equiv ==> equiv ==> equiv) choose_new_pos}.

(* La position choisie est a moins de Dp du robot pris pour cible, et
est atteignable en un round. *)
Axiom choose_new_pos_correct : ∀ obs target new,
    new ≡ choose_new_pos obs target →
    (dist new target ≤ Dp ∧ dist new (0,0) ≤ D)%R.
```

Figure 7.6: Les définitions et propriétés des fonctions `move_to` et `choose_new_pos` utilisées dans le robogram.

Définition 7.2.1 (Pas de collisions) *: pour tout robot, au début de la phase* Move*, il n'existe aucun robot à moins de D.*

Le prédicat pour les collisions est simple, si un robot est à moins de D de nous, il est possible de le percuter en une phase de *Move*, et donc il faut qu'aucun des robots ne soit à moins de D. Ce prédicat est transcrit comme suit dans le code Coq:

```
Definition no_collision_conf (conf : config) :=
(* Pour tous robots distincts *)
∀ g g', g ≠ g' →
  (* si ces deux robots sont vivants *)
  get_alive (conf (Good g)) = true → get_alive (conf (Good g')) = true →
  (* ils ne sont jamais a la meme position *)
  dist (get_location (conf (Good g))) (get_location (conf (Good g'))) ≠ 0ℝ.
```

Définition 7.2.2 (Chemin) *: pour tout robot, il existe un chemin entre lui et le robot libre dans le graphe de visibilité.*

Tous les robots doivent être reliés au robot libre. Pour les besoins de la preuve, nous avons transformé ce prédicat en un prédicat équivalent `path_conf` énoncé ci-dessous, qui est basé sur le fait que le robot libre a le plus petit identifiant.

Définition 7.2.3 (Cible) *: pour tout robot autre que le robot libre, il existe un robot d'identifiant inférieur visible.*

```
Definition path_conf (conf:config) :=
(* Pour tout robot vivant *)
∀ g, get_alive (conf (Good g)) = true →
          (* soit ce robot est le robot libre *)
          (get_ident (conf (Good g)) = 0 ∨
          (* soit il existe un autre robot vivant *)
          (∃ g', get_alive (conf (Good g')) = true ∧
          (* ce robot etant a moins de Dmax, *)
          (dist (get_location (conf (Good g))) (get_location (conf (Good g'))) ≤
Dmax)ℝ ∧
          (* et d'identifiant inferieur. *)
          get_ident (conf (Good g')) < get_ident (conf (Good g)))).
```

Pour prouver ces prédicats nous définissons une configuration de départ respectant des contraintes simples (7.7):

- Il n'existe aucun robots à moins de D d'un autre robot (`config_init_not_close`),
- Tous les robots ont leur lumière éteinte (aucun n'est en danger) (`config_init_off`),
- Pour tout robot autre que le robot libre, il existe un robot d'identifiant inférieur à moins de $Dmax$ (`config_init_lower`).

Les prédicats sont trivialement vérifiés sur cette configuration.

Nous voulons maintenant montrer que ces prédicats perdurent tout au long de l'exécution. Mais ces prédicats seuls ne suffisent pas. Nous rajoutons donc des propriétés qui, avec les prédicats de départ, permettent de former un ensemble de propriétés perdurant tout au long de l'exécution. Ces propriétés sont:

```
(* Tous les robots sont a plus de D dans la conf init *)
Definition config_init_not_close (config_init:config) := ∀ id,
    match (config_init id) with
      (pos, (((ident, light), alive),based)) ⇒
      based = false →
      ∀ id',
      match config_init id' with
        (pos', (((ident', light'), alive'), based')) ⇒
               based' = false → (dist pos pos' > D)ℝ end end.

(* Tout robot est eteint au debut *)
Definition config_init_off (config_init:config) := ∀ id,
    match (config_init id) with
      (_,((( _, light), _),_)) ⇒ light = false
    end.

(* Tout robot voit un robot inferieur, ou alors est le robot libre. *)
Definition config_init_lower (config_init:config) :=
∀ id,
    match (config_init id) with
      (pos,(((ident,light),alive), based))
      ⇒ alive = true →
         ident = 0 ∨
         ∃ id', match(config_init id') with
                       (pos',(((ident',_),alive'), _)) ⇒
                       alive' = true ∧
                       ident' < ident ∧
                       (dist pos pos' < Dmax)ℝ
                                          end
      end.

(* Le predicat regroupant les trois caracteristiques de la
   configuration de base *)
Definition config_init_pred config :=
  config_init_lower config
  ∧ config_init_off config
  ∧ config_init_not_close config.
```

Figure 7.7: Le prédicat définissant une configuration initiale valide, ainsi que les définitions de ses composantes.

```
Definition executed_means_light_on conf := ∀ g da,
    da_predicat da →
    get_alive (conf (Good g)) = true →
    get_alive (round rbg_ila da conf (Good g)) = false →
    get_light (conf (Good g)) = true.

Definition executioner_means_light_off conf :=
  ∀ g da,
    da_predicat da →
    get_alive (conf (Good g)) = true →
    (∃ g', get_alive (conf (Good g')) = true ∧ get_based (conf (Good g')) = false ∧
              get_ident (conf (Good g)) < get_ident (conf (Good g')) ∧
              (dist (get_location (conf (Good g))) (get_location (conf (Good g')))
                   ≤ D)ℝ) →
    get_light (conf (Good g)) = false.

Definition  exists_at_less_that_Dp conf :=
  ∀ g,
    get_alive (conf (Good g)) = true →
    get_ident (conf (Good g)) > 0 →
    (∀ g_near, get_alive (conf (Good g_near)) = true →
                  (dist (get_location (conf (Good g_near)))
                        (get_location (conf (Good g))) ≤ Dmax)ℝ →
                  get_ident (conf (Good g_near)) < get_ident (conf (Good g)) →
                  get_light (conf (Good g_near)) = true) →
    ∃ g',
      get_alive (conf (Good g')) = true ∧
      get_ident (conf (Good g')) < get_ident (conf (Good g)) ∧
      (dist (get_location (conf (Good g))) (get_location (conf (Good g'))) ≤
Dp)ℝ.
```

Figure 7.8: Le code COQ des prédicats utilisés dans la preuve.

`executed_means_light_on` : un robot s'éliminant a forcément sa lumière allumée.

`executionner_means_light_off` : un robot éliminant un autre robot a sa lumière éteinte.

`exists_at_less_that_Dp` : si un robot ne voit que des robots à lumière allumée, un de ces robots est au plus à Dp.

Les définitions COQ de ces prédicats sont disponibles à la figure 7.8.

Nous montrons maintenant qu'avec ces propriétés ajoutées aux prédicats, le tout est préservé d'un round au suivant.

7.2.3 no_collision

Nous voulons prouver que deux robots différents et vivants n'ont jamais la même position tout au long de l'exécution. Cette propriété étant vérifiée dans la configuration de départ, il reste donc à prouver que ce prédicat est préservé par la fonction de transition. Pour faire cela, nous commençons dans une configuration $conf$ où tous les prédicats sont vérifiés.

Nous voulons prouver que la propriété de non collision est respectée après le round. Nous prouvons donc que s'il devait y avoir une collision, alors il y aurait une contradiction.

Pour cela, nous commençons par comparer les positions de deux robots `g` et `g'` dans la configuration actuelle et dans la configuration après un round.

- Le cas où rien ne bouge est trivial, car la configuration actuelle respecte `no_collision_conf`.
- Si un seul robot bouge et qu'il était le robot d'identifiant inférieur, alors l'autre robot aurait dû s'éliminer si une collision était possible, la collision est donc impossible.
- Si c'est le robot d'identifiant supérieur qui bouge, alors il y a contradiction car il est trop près du robot inférieur pour bouger.
- Si les deux robots bougent, là encore le robot d'identifiant supérieur ne devrait pas pouvoir bouger car il existe un robot inférieur à lui à moins de $2 \times D$.

Nous pouvons donc prouver le lemme suivant:

```
Lemma no_collision_cont : ∀ (conf:config) (da:da_ila),
no_collision_conf conf → da_predicat da → path_conf conf →
no_collision_conf (round rbg_ila da conf).
```

7.2.4 executed_means_light_on

Nous allons maintenant nous attaquer au deuxième prédicat, à savoir qu'un robot se faisant éliminer par un robot d'identifiant inférieur trop près a forcément sa lumière allumée.

La propriété correspondante est :

```
Definition executed_means_light_on conf := ∀ g da,
    da_predicat da →
    get_alive (conf (Good g)) = true →
    get_alive (round rbg_ila da conf (Good g)) = false →
    get_light (conf (Good g)) = true.
```

Pour cela, nous montrons qu'il est impossible qu'un robot s'élimine si sa lumière n'est pas allumée. Nous établissons donc la propriété suivante :

```
Lemma executed_means_light_on_round :
∀ conf da,
    (* si l'action demoniaque respecte nos hypotheses, *)
    da_predicat da →
    (* que chaque robot voit un robot d'identifiant inferieur, *)
    path_conf conf →
    (* qu'aucun robot n'est a moins de D d'un autre robot, *)
    no_collision_conf conf →
    (* que tout robot provoquant l'elimination d'un autre robot a sa
    lumiere eteinte, *)
    executioner_means_light_off conf →
    (* que tout robot s'eliminant a sa lumiere allumee*)
    executed_means_light_on conf →
    (* et que tout robot voit un robot a moins de Dp de lui *)
    exists_at_less_that_Dp conf →
    (* alors au round suivant, les robots s'eliminant auront toujous
    tous leur lumiere allumee. *)
    executed_means_light_on (round rbg_ila da conf).
```

Dans cette partie et les suivantes, nous nommons la configuration de base C, la configuration après un round C_r et la configuration après deux round C_{2r}. Le robot g est celui auquel nous nous intéressons.

Nous savons que g est vivant lors de C_r, de par la définition d'`executed_means_light_on` pour C_r. De même, g est éliminé dans C_{2r}. Or, si un robot s'élimine, soit aucun robot n'est à portée, soit un robot est trop proche. Pour que la lumière de g soit allumée, il a fallu que la fonction `move_to` lors du passage de C à C_r ait renvoyé `false`. Nous allons donc montrer que le cas contraire est impossible: si g peut bouger normalement, c'est-à-dire si aucun robot ne le gène, il ne peut pas s'éliminer.

Si entre la configuration C et C_r, g ne bouge pas, alors dans la configuration C, il n'y a personne à moins de $2 \times D$ de lui, et au moins un robot à au plus Dp de lui, grâce aux propriétés sur `move_to` et `chose_new_pos`. Au round suivant, il n'existe donc personne à moins de D de lui et au moins un robot à moins de $Dmax$. Le robot g ne s'élimine donc pas dans C_{2r}. Il y a contradiction.

Si g bouge entre C et C_r, alors comme g s'élimine dans C_{2r}, nous utilisons un raisonnement similaire : si g meurt car aucun robot n'est à portée, il y a un problème avec le fait que `choose_new_pos` renvoie une position à moins de Dp de la cible et si g meurt car il y a un autre robot trop proche, de la même façon, `move_to` renvoyant `true` pose une contradiction.

Nous avons donc prouvé qu'après un round la propriété `executed_means_light_on` était préservée.

Cette preuve COQ fait 2300 lignes dans notre développement formel.

7.2.5 executioner_means_light_off

Nous allons maintenant prouver que tout robot faisant s'éliminer un autre robot (et que nous appellerons donc un *bourreau*) a forcément sa lumière éteinte. Nous traduisons dans un premier temps cette propriété en COQ:

```
Definition executioner_means_light_off conf :=
  ∀ g da,
    da_predicat da →
    get_alive (conf (Good g)) = true →
    (∃ g', get_alive (conf (Good g')) = true ∧
                get_ident (conf (Good g)) < get_ident (conf (Good g')) ∧
                (dist (get_location (conf (Good g))) (get_location (conf (Good g')))
                     ≤ D)%R) →
    get_light (conf (Good g)) = false.
```

La preuve de préservation de ce prédicat est relativement simple quand nous avons la preuve qu'un robot changeant de position a toujours sa lumière éteinte, c'est-à-dire que son paramètre `light` vaut `false`.

Commençons par prouver cette dernière affirmation.

```
Lemma moving_means_light_off : ∀ conf g (da:da_ila),
    da_predicat da →
    get_alive (round rbg_ila da conf (Good g)) = true →
    get_location (conf (Good g)) ≠ get_location (round rbg_ila da conf (Good g)) →
    get_light (round rbg_ila da conf (Good g)) = false.
```

Le robot g est vivant après un round, par définition, donc nous pouvons regarder ce que renvoie la fonction `move_to`.

- Si elle renvoie `true`, alors le robogram met la lumière à `false` et la propriété est validée.
- Si la fonction renvoie `false`, alors le robogram ne change pas la position du robot g, ce qui rentre en contradiction avec les hypothèses.

Revenons à la preuve qu'un bourreau, c'est-à-dire un robot qui provoque l'élimination d'un autre robot, a toujours sa lumière éteinte.

Nous commençons par montrer que le robot bourreau b bouge. Par définition, il existe un robot g à une distance inférieure à D de b dans la configuration C_{2r}, donc g mourra lors du passage à la prochaine configuration. Il est donc possible d'utiliser le résultat précédent de `executed_means_light_off_round` pour montrer que la lumière de g était allumée. Donc, par définition du robogram, la lumière de g est allumée à la configuration C_r, donc g n'a pas bougé entre les configurations C et C_r. Or, par définition de `executioner_means_light_off`, g était vivant dans C_r. Donc b a forcément bougé entre C et C_r. En utilisant `moving_means_light_off`, la lumière de b est éteinte dans C.

7.2.6 exists_at_less_that_Dp

Ce prédicat énonce que pour tout robot g, si tous les robots d'identifiant inférieur visibles sont allumés, alors il existe un de ces robots à moins de Dp.

```
Definition  exists_at_less_that_Dp conf :=
  ∀ g,
    get_alive (conf (Good g)) = true →
    get_ident (conf (Good g)) > 0 →
    (∀ g_near, get_alive (conf (Good g_near)) = true →
                    (dist (get_location (conf (Good g_near)))
                          (get_location (conf (Good g))) ≤ Dmax)ℝ →
                   get_ident (conf (Good g_near)) < get_ident (conf (Good g)) →
                   get_light (conf (Good g_near)) = true) →
    ∃ g',
      get_alive (conf (Good g')) = true ∧
      get_ident (conf (Good g')) < get_ident (conf (Good g)) ∧
      (dist (get_location (conf (Good g))) (get_location (conf (Good g'))) ≤ Dp)ℝ.
```

Pour prouver le fait que cette propriété est préservée par la fonction `round` nous allons commencer par remarquer qu'il existe forcément un robot à moins de $Dmax$ de g dans C_r car g est vivant. Nous regardons donc le choix de cible à partir de la configuration C grâce à `choose_target`: si ce robot est à moins de Dp, et qu'il est allumé, il ne bougera pas. De plus, comme tous les robots visibles sont allumés, aucun autre robot ne peut s'approcher assez pour l'éliminer, donc ce robot sera celui à moins de Dp. Si la cible choisie est à plus de Dp, alors il n'existe aucun robot à moins de Dp grâce aux propriétés sur le choix de la cible. Il y a donc un problème. Montrons maintenant la contradiction.

Pour cela, nous allons regarder le choix de cible précédent, celui depuis la configuration C. Si la cible choisie était allumée, alors grâce à la propriété `exists_at_less_that_Dp`, il y avait au moins un robot à moins de Dp, donc la cible choisie, t, était elle aussi à moins de Dp. La position de g dans C_r est donc à moins de Dp de t, que g bouge ou non. Le robot t est donc à moins de Dp de g dans C_r, ce qui contredit le fait que la cible choisie depuis C_r est, elle, a plus de Dp.

Nous avons donc en Coq la propriété suivante:

```
Lemma  exists_at_less_round :   ∀ conf da,
    da_predicat da →
    path_conf conf →
    no_collision_conf conf →
    executioner_means_light_off conf →
    executed_means_light_on conf →
     exists_at_less_that_Dp conf →
     exists_at_less_that_Dp (round rbg_ila da conf).
```

7.2.7 path_conf

Avec toutes ces propriétés, il est maintenant possible de prouver la préservation pour tout robot r de l'existence d'un robot r' d'identifiant inférieur à au plus $Dmax$ de r lors de l'exécution:

On note pour tout robot X, $loc(X)$ sa position, et $loc_r(X)$ sa position après un round.

Un robot r choisit un robot cible c, il y a deux possibilités:

1. Soit il existe une position acceptable à moins de Dp de $loc(c)$, sans robot aux alentours, et dans ce cas là, r se déplace vers $loc(c)$,
2. soit r ne peut pas bouger car un robot g le gêne.

Si r se déplace

Il reste à étudier les cas selon l'état de c:

- Si c est éteint, il ne s'élimine pas grâce à la propriété `executed_means_light_on`, comme r bouge vers $loc(c)$ suffisamment pour que la distance soit au plus Dp, la distance entre $loc_r(r)$ et $loc_r(c)$ est au plus $Dmax$.
- Si c est allumé, il risque de s'éliminer. Selon les règles que nous nous sommes imposées dans la partie 7.1.2 pour choisir une cible, si c est allumé, alors tous les robots visibles le sont. De plus, par la propriété `exists_at_less_that_Dp`, il en existe au moins un tel que sa position soit à moins de Dp. En utilisant encore les règles de choix de la cible, c est alors aussi à moins de Dp. Comme tous les robots sont allumés, il ne peut pas y avoir de robot provoquant l'élimination d'un autre robot, ceux-ci étant forcément éteints par la propriété `executioner_means_light_off`. Il n'existe donc pas de bourreau pour c, qui sera vivant au round suivant, et sera un robot à moins de $Dmax$ de r.

Si r est gêné par g

Il n'est pas possible de se déplacer vers c à cause du robot gênant g, donc $loc(g)$ est à moins de $3 \times D$ de $loc(r)$, étant à moins de $2 \times D$ d'une position à moins de D de $loc(r)$. Dans ce cas là, $loc(r) = loc_r(r)$, et nous faisons attention à l'état de g. Si g ne s'élimine pas, $loc_r(g)$ est à au plus $3 \times D + D$ de $loc(r)$, donc à moins de $Dmax$. Si g s'élimine, il existe donc un robot bourreau b, avec $dist\ loc(b)\ loc(g) \leq D$. Or, comme b est un bourreau, il doit avoir sa lumière éteinte en respectant `executioner_means_light_off`. Ainsi b est éteint et ne s'éliminera pas au tour prochain, ce qui le rend éligible à être le robot vivant à moins de $Dmax$ de $loc_r(r)$.

Nous avons donc prouvé que l'existence d'un chemin était préservée si les autres propriétés l'étaient.

```
Lemma validity:
(* pour tout demon et toute configuration de depart *)
∀(demon : demon_ila_type) conf,
(* si la configuration de depart est une configuration valide *)
  conf_pred conf →
(* et que le demon est bien defini *)
  demon_ILA demon →
(* alors il n'y a pas de collision et le chemin entre tout robot et le
robot cible est preserve*)
  NoCollAndPath (execute rbg_ila demon conf).
```

Nous avons donc le théorème suivant: en partant d'une configuration initiale correcte, les prédicats d'existence du chemin et de non collision entre les robots sont préservés.

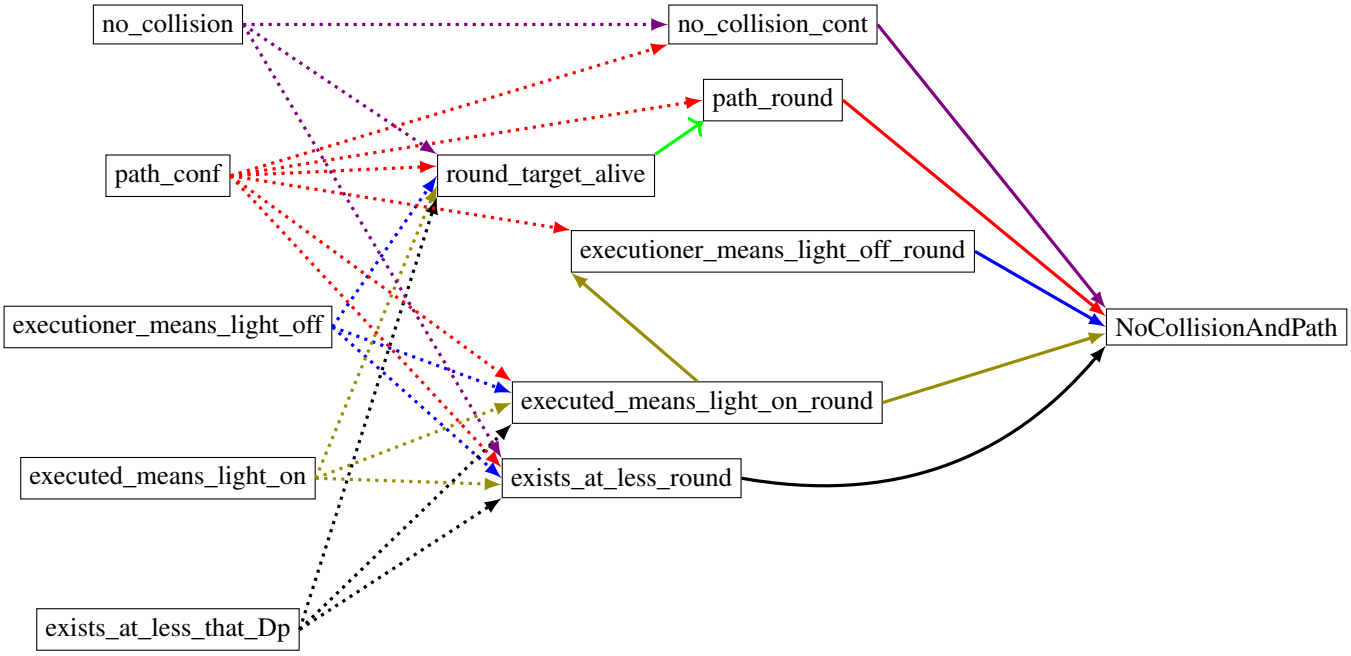


Figure 7.9: Représentation des dépendances pour la preuve. Les flèches pointillées représentant les dépendance des prédicats sur une configuration, alors que les flèches pleines sont les dépendance des fonction entre deux configuration successives. Les couleurs représente quel prédicat est prouvé.

```
Parameter config_init: config.
Axiom config_init_pred_true : config_init_pred config_init.

Theorem validity_conf_init:
  ∀ demon,
  demon_ILA demon →
  NoCollAndPath (execute rbg_ila demon config_init).
```

7.3 Une base fixe à relier à la cible

Afin de représenter une station de secours devant être reliée à une équipe de recherche, nous ajoutons une base fixe comportant un réservoir de robots inactifs. Cette base active un robot dès qu'il y a un risque que le robot libre (par exemple l'équipe de recherche) ne soit plus relié.

Nous posons les hypothèses que :

- Les robots de la base ont tous un identifiant supérieur à ceux des robots actifs en dehors de la base et
- Le robot sortant de la base est aussi le robot avec le plus petit identifiant de la base.
- La base se trouve en $(0, 0)$ dans le référentiel global.

7.3.1 Ajouts dans les définitions

Nous ajoutons aux informations du robot un paramètre supplémentaire : un booléen indiquant si le robot est à la base ou non. Nous créons la fonction de projection *get* correspondante, et modifions les autres fonctions d'accès pour qu'elles correspondent au nouveau format d'information :

```
Definition based := bool.
...
Definition ILA :Type := identifiants*light*alive*based.
...
Definition config:= @configuration Loc (R2*ILA) State_ILA Identifiers.
...
Definition get_based (elt : R2*ILA) : alive := snd(snd(elt)).
```

Un robot ne peut pas sortir de la base à n'importe quel moment : trop tôt, il pourrait y avoir des risques de collisions si un autre robot déjà hors de la base était encore trop près. On interdit donc à un robot de sortir si il y a un robot à moins de $2 \times D$ de la base.

Un robot ne doit pas sortir trop tard non plus au risque de voir la base n'être plus reliée au reste des robots.

À cause des risques d'élimination et des robots bougeant entre les rounds, tout protocole doit faire sortir un robot r_b de la base dès que tous les robots vivants sont à plus de $Dp - 3 \times D$. Si le protocole attend plus, le robot r qui était entre $Dp - 3 \times D$ et $Dp - 2 \times D$ de la base au round précédent sera potentiellement entre $Dp - 2 \times D$ et $Dp - D$. Ce robot r étant potentiellement en danger, il risque de s'éliminer à ce round, et ce à cause d'un robot b qui lui est entre $Dp - D$ et Dp de la base. Le robot b peut bouger pendant le round où r_b sort de la base. Ainsi au premier round où r_b fait son snapshot hors de la base, b peut être le seul robot visible par r_b et être à plus de Dp. On se retrouve avec un cas invalidant le prédicat `exists_at_less_that_dp`.

En effet, Pour les mêmes raisons que ci-dessus, si le robot r_b sortant de la base est a $Dmax - D$ d'un robot b à portée, il est possible que b soit éliminé ce même round. Ainsi, le robot c bourreau de b est potentiellement à $Dmax$ lorsque qu'il choisit de bouger et donc est hors de portée de r_b au round suivant.

Cette limite impose aussi d'augmenter la valeur minimale de $Dmax$ à $7 \times D$, car $Dp - 3 \times D = Dmax - 4 \times D$, et donc si $Dmax = 6 \times D$, un robot ne pourrait sortir que lorsqu'il serait déjà trop tard.

La fonction `update` est modifiée en conséquence pour ajouter la base dans le modèle (voir figure 7.10).

7.3.2 Les prédicats ajoutés

Les robots de la base ont des propriétés exprimées par des prédicats adaptés.

- Un robot à la base est vivant et en position $(0, 0)$ dans le référentiel global.

```
(∀ g, get_based (conf (Good g)) = true →
       get_alive (conf (Good g)) = true
        ∧ get_location (conf (Good g)) = (0,0))
```

- La base est reliée au reste des robots, c'est-à-dire qu'à tout moment, il existe un robot assez près de la base.

```
(∃ g, get_based (conf (Good g)) = false
          ∧ get_alive (conf (Good g)) = true
          ∧ (dist (get_location (conf (Good g))) (0,0)ℝ ≤ Dp - D)
```

- Un robot à la base est plus grand que tout autre robot hors de la base.

```
    (* [too_far_aux1 c g g'] renvoie [true] si le robot g a un
       identifiant strictement plus petit que g' *)
Definition too_far_aux1 config g g' :=
   ((if (negb (Nat.eqb (get_ident (config (Good g))) (get_ident
   (config (Good g'))))))
         then
         ((Nat.ltb (get_ident (config (Good g)))
                   (get_ident (config (Good g'))))))
          else true)).

(* [too_far_aux2 c g g'] renvoie [false] si le robot g' est vivant, hors de
la base  et a moins de [Dp-3*D] de g *)
Definition too_far_aux2 config g g' :=
    (negb (andb (negb (get_based (config (Good g'))))
                (andb (get_alive (config (Good g')))
                      (Rle_bool (dist (get_location (config (Good g')))
                                      (get_location (config (Good g))))
                                (Dp-3*D)ℝ)))).

(* Si un robot est au niveau de la base *)
Definition upt_robot_too_far config g fchoice:=
  if R2dec_bool (retraction fchoice (get_location (config (Good g))))
  (0,0)ℝ
(* alors si grace a la premiere fonction, on sais que tous les autres
robots a la bases sont d'identifiant plus petits *)
  then if (forallb (too_far_aux1 config g)
           (List.filter (fun g' ⇒ get_based (config (Good g'))
                                && get_alive (config (Good g')))
                                Gnames))
(* et que tous les autres robots vivants sont a au moins [DP-3*D] et
   hors de la base*)
        then if (forallb (too_far_aux2 config g) Gnames)
        (* alors on renvoie [true], sinon on renvoie [false] *)
        then true else false
       else false
  else false.


Definition upt_aux (config:config) (g: G) (rl : R2*light) fchoice: (R2*ILA) :=
  match config (Good g) with (r,(((i,l),a),b)) ⇒
  (* La fonction upt_robot_too_far est appliquee a tout robot etant a la base*)
  if b then
    (* si elle renvoie [true], c'est-a-dire que les autres robots de
    la base sont plus petits et qu'il n'y a pas de robot vivant pres
    de la base, *)
    if upt_robot_too_far config g (frame_choice_bijection fchoice) then
      (* alors le robot choisi sort de la base *)
      (r, (((i,false),true),false))
    else
      (* sinon l'etat du robot ne change pas *)
      config (Good g)
  else
    (* sinon on fait comme precedemment *)
    if upt_robot_dies_b config g
    then (r,(((i,l),false),b))
    else (fst rl,(((i,snd rl), a),b))
  end.
```

Figure 7.10: Les fonctions modifiées et ajoutées pour la prise en compte de la base, en rapport avec la fonction `update`.

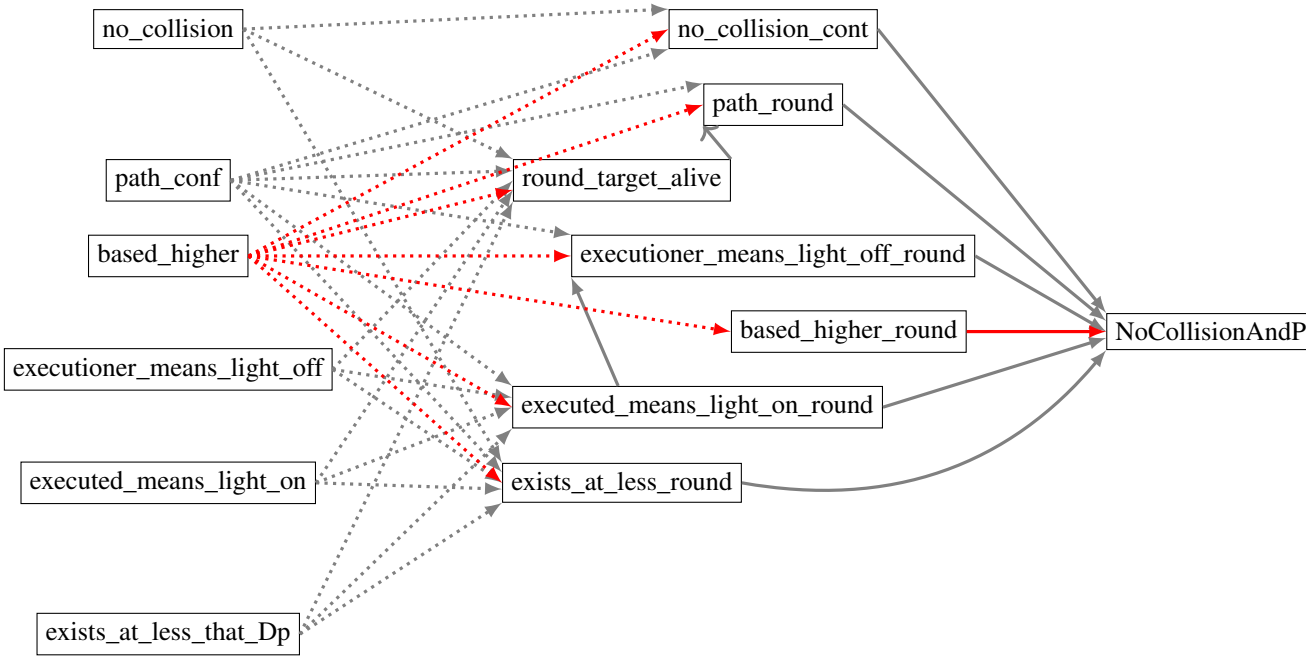


Figure 7.11: Représentation des dépendances pour la preuve avec base fixe, les dépendance existant dans la preuve précédente restent les mêmes, nous avons ajouté en rouge les dépendances venant des prédicats liés à la base.

```
(∀ g g', get_based (conf (Good g)) = true →
        get_based (conf (Good g')) = false →
        get_ident (conf (Good g')) < get_ident (conf (Good g))
```

Ces trois prédicats sont regroupés en un seul grâce à une conjonction. Ce prédicat, nommé `based_higher`, ajouté, nous avons dans un premier temps utilisé les prédicats précédents pour prouver sa validité durant toute l'exécution. Cependant il est nécessaire de rajouter une propriété pour s'assurer de cette validité : il faut qu'il existe au moins un robot à la base. Cette propriété n'est pas mise en axiome car elle rentrerait en contradiction avec le fait que PACTOLE n'utilise actuellement qu'un nombre fini de robots.

```
Lemma based_higher_round : ∀ conf da,
    da_predicat da →
    path_conf conf →
    based_higher conf →
    executioner_means_light_off conf →
    executed_means_light_on conf →
    (∃ g, get_based (conf (Good g)) = true) →
    based_higher (round rbg_ila da conf).
```

Les autres preuves ont aussi été modifiées pour refléter les modifications apportés. Nous arrivons à la figure 7.11.

Nous avons donc le théorème final: en partant d'une configuration initiale valide, et pour tout démon répondant à nos caractéristiques, tant qu'il existe un robot à la base, en utilisant le protocole décrit, aucune collision n'aura lieu, et tout les robots auront le robot libre dans leur graphe de visibilité.

En particulier : le robot libre est toujours dans le graphe de visibilité de la base.

```
Parameter R_r : R.

Axiom R_r_0: (R_r > 0)%R.
```

Figure 7.12: La déclaration du rayon, et les propriété nécessaire le concernant.

```
Theorem validity_conf_init:
   (* pour tout demon respectant nos contraintes*)
   ∀ demon, demon_ILA demon →
   (* s'il existe un robot a la base *)
   exists_at_based (execute rbg_ila demon config_init) →
   (* alors il n'y aura aucune collision et les robots seront lies au
   robot libre durant l'execution.*)
   NoCollAndPath (execute rbg_ila demon config_init).
```

7.4 Des robots avec un volume

Pour maintenant être encore plus proche de la réalité physique des robots, nous voulons représenter les robots non plus comme des points, mais comme des boules d'un certain rayon que nous noterons `R_r`. Cette boule est une approximation des volumes réels des robots, mais est définie de sorte que le volume y soit inscrit.

Définition 7.4.1 (Le rayon R_r) *Le rayon* `R_r` *est le rayon définissant la boule autour du robot à partir de laquelle un autre robot rentre en contact.*

Nous avons bien sûr laissé ce rayon abstrait, il est néanmoins nécessaire d'avoir des hypothèses sur ce rayon (voir figure 7.12).

Il nous faut maintenant mettre à jour les preuves pour que les distances reflètent ce rayon supplémentaire pour chaque robot. Il faut aussi réévaluer notre hypothèse sur `Dmax` car maintenant un robot doit voir plus loin pour que toutes les propriétés soient préservées.

Dès qu'un robot doit regarder si un autre robot est assez près pour être dangereux, il faut rajouter une fois le rayon pour la perception du robot qui regarde, et une autre fois le rayon pour le robot qui est observé. Par exemple, la fonction qui regarde si un robot doit s'éliminer change comme suit:

```
Definition upt_robot_dies_b (config:config) (g:G) : bool :=
  existsb (fun elt : R2 * ILA ⇒ Rle_bool (dist (@get_location Loc _ _ elt)
 (* la distance minimale est augmentee de 2 fois le rayon *)
  (@get_location Loc _ _ (config (Good g)))) (D+2*R_r)%R)
          (List.filter (fun (elt : R2*ILA) ⇒ (get_ident(elt) <?
          get_ident (config (Good g)))
          && get_alive (elt) && (negb (get_based (elt)))) (config_list config))
  || negb
  (existsb (fun elt : R2 * ILA ⇒ (Rle_bool (dist (@get_location Loc _ _ elt)
  (* la distance maximale n'est pas augmentee elle car elle est redefinie *)
     (@get_location Loc _ _ (config (Good g)))) Dmax))
          (List.filter (fun (elt : R2*ILA) ⇒ (get_ident(elt) <? get_ident (config (Good g)))
          && get_alive (elt) && (negb (get_based (elt))))(config_list config))).
```

Ainsi, les distances relatives augmenteront, ce qui modifiera notr hypothèse sur distance maximale de vision des robots comme suit:

```
Parameter Dmax : R.
Axiome Dmax_7D : Dmax > 7*D + 6*R_r
```

Cette distance s'explique de la même manière que nous avions justifié que sans les volumes, la distance maximale était au minimum de $7 \times D$ mais pour chaque fois qu'un robot doit regarder si un autre robot est potentiellement trop près, il faut rajouter `R_r` pour les deux robots.

Le principe des preuves lui ne change pas, juste les calculs nécessaires. La preuve du fait que notre protocole évite les collisions entre les robots, et qu'il permet le maintien d'une connexion entre la base et le robot libre, est toujours valide, même en prenant en compte les volumes des robots.

Chapter 8

Conclusion

Nous nous sommes intéressés à l'enrichissement d'une bibliothèque formelle afin d'offrir les moyens de garantir résultats théoriques et comportements de protocoles distribués dans un modèle émergent : les essaims de robots mobiles.

En considérant un modèle à l'énoncé simple mais présentant une grande diversité d'ajustements possibles, nous nous sommes appuyés sur un prototype pour la preuve formelle dans ce cadre : la bibliothèque PACTOLE[1].

Le prototype précédant nos travaux reste essentiellement proche de la preuve de concept. Il permet toutefois d'exprimer des systèmes de robots évoluant dans des espaces continus, de type plan réel $\mathbb{R}^2$, pour des démons totalement ou semi-synchrones, avec des déplacements rigides ou flexibles.

Ce sont des possibilités suffisantes pour garantir ou établir certains résultats d'impossibilité ou de correction de protocoles. Ce sont néanmoins des limitations dans l'usage qu'on veut courant et aisé de la vérification, en particulier dans la perspective d'aborder des cadres et applications réalistes des essaims de robots.

Il fallait donc à ce stade des extensions du cadre formel, d'une part, à des préoccupations non abordées mais pourtant courantes dans la littérature sur les essaims de robots mobiles, à savoir les problématiques sur les graphes, et, d'autre part, à des contextes plus réalistes : dans les espaces, dans les caractéristiques des robots, dans la caractérisation du synchronisme des exécutions.

Contributions

Nous proposons des évolutions majeures du cadre formel correspondant à ces attentes.

Dans un premier temps, nous permettons la prise en compte des graphes : nous définissons classes et interfaces pour une expression de ces espaces aisée et suffisamment simple pour qu'un non spécialiste de la preuve formelle puisse les aborder.

Il ne suffit pas de pouvoir exprimer un objet, il faut bien sûr que celui-ci soit effectivement manipulable dans un développement formel. Nous illustrons l'utilisabilité de notre extension en donnat la première preuve formelle de l'impossibilité de l'exploration avec stop dans un anneau quand le nombre de robots divise la taille de l'anneau. Ces premières contributions sont publiées dans les actes de la conférence internationale ICDCN2018 [12].

Dans un deuxième temps, nous abordons le type de synchronisation le plus complexe et le plus général : le mode asynchrone. Étendre le cœur du modèle aux exécutions asynchrones entraîne une généralisation de la plupart des constructions existantes dans la bibliothèque formelle. De nouveau, nous montrons que notre extension est utilisable. Nous prouvons formellement une équivalence en mode asynchrone entre

[1] https://pactole.liris.cnrs.fr

deux variantes de graphes, graphes discrets d'une part, graphes continus (c'est-à-dire où les agents se déplacent continûment le long des arêtes) d'autre part mais avec détection dans le voisinage des sommets. Cette preuve utilise bien sûr notre modélisation des graphes. Ces travaux sont présentés et publiés sous forme courte dans les actes de la conférence SSS2018 [7] et sous forme normale dans les actes des conférences NETYS2019 [8] et ALGOTEL2020 [9].

Enfin, dans un troisième temps, nous explorons toujours plus de réalisme et travaillons à la représentation des robots présentant un volume et, donc, des risques de collision. Nous développons et exprimons dans le cadre formel étendu à dessein un protocole permettant de maintenir, par un essaim de robots mobiles, un chemin de connexions (à portées limitées) entre une base fixe et une cible mobile. Nous établissons formellement que notre protocole garantit le maintien de la connexion entre la base et la cible à tout point d'une exécution synchrone et sans collision.

Nos travaux développés initialement dans une version de PACTOLE à base de modules sont intégrés à la version courante à base de typeclasses.

Perspectives

Les perspectives d'évolutions et d'extensions du cadre formel sont nombreuses, à différents termes et sous différents aspects.

Du point de vue de la modélisation. Diverses familles ou sous-familles d'espaces peuvent être modélisées à court terme. Les espaces tridimensionnels sont bien sûr d'une grande importance pour toutes les problématiques liées aux drones aériens ou sous-marins, etc. Comme graphes particuliers les grilles (finies ou infinies) présentent aussi un contexte intéressant avec des études de cas issues de nouveaux protocoles : on pourrait par exemple certifier formellement le très récent résultat de Bramas et al. établissant que seulement 5 robots lumineux avec chiralité sont nécessaires et suffisants pour explorer une grille infinieé[22].

À plus long terme on peut signaler deux problématiques fondamentales.

La première concerne les *fautes byzantines*. Des travaux abordant la certification en présence de robots byzantins ont déjà été certifiés à l'aide de PACTOLE. Ils ne concernent cependant qu'une preuve d'impossibilité, fondée sur le codage d'un contre-exemple. Le véritable défi à relever serait de donner la possibilité de preuve formelle de correction et résilience d'un protocole. Au delà de la modélisation de fautes et comportements byzantins, il faut savoir rendre la preuve *praticable*, c'est-à-dire veiller à ce que la modélisation des fautes soit utilisable effectivement dans un développement, sans introduire de niveaux de difficulté et complexité insurmontables.

La deuxième s'intéresse aux *comportements probabilistes*, que ce soit du côté robot ou du côté environnement.

Du côté robot, l'utilisation d'algorithmes randomisés est bien utile pour briser des symétries rendant inopérants des protocoles complètement déterministes : il a par exemple été prouvé dans la section 5 que l'exploration avec stop était impossible si le nombre de robots divisait la taille de l'anneau mais pour des protocoles *déterministes* ; Devismes et al. proposent un protocole probabiliste ne nécessitant que 4 robots (nombre nécessaire et suffisant) pour l'exploration de tout anneau de taille $n > 4$ [37].

Du côté environnement, on veut pouvoir envisager des choix démoniaques contrôlés par une certaine distribution.

Les travaux dans ce cadre devront s'appuyer sur des bibliothèques formelles dédiées (comme `alea` [5] ou plus récentes comme `polaris` [2] L'inclusion d'information de mesures risque évidemment d'entraîner une restructuration du cœur du modèle.

[2] https://github.com/jtassarotti/polaris

Du point de vue de l'aide au développement et à la vérification. Proposer un environnement d'aide au développement sûr de protocoles pour les essaims de robots et à la vérifications formelles de résultats théoriques dans ce modèle n'a de sens que si cet environnement s'adresse à tous, et pas seulement aux spécialistes de preuve formelle. L'accent doit donc toujours être mis sur l'aisance d'utilisation et en particulier la facilité de *spécification*.

Il importe donc de savoir mettre à disposition les moyens de créer et d'instancier les modèles de la manière la plus efficace et automatique possible. C'est une tâche au long cours. Le besoin à court terme est de proposer de façon automatisée en passant par des fichiers de configuration simples des initialisations de modèles quasi automatiques. La mise en place du contexte de travail est en effet encore trop complexe dans ce type de prototype pour une véritable démocratisation de leur utilisation.

L'automatisation a aussi son rôle à jouer dans l'aide à la preuve. L'évolution de cadres formels de type PACTOLE devrait donc inclure des procédures de (semi-)décision déchargeant l'utilisateur de preuves fastidieuses ou techniques dans des cas bien définis. On peut en particulier penser à des topologies particulières se prêtant plutôt bien à cette automatisation comme des graphes particuliers.

Ces perspectives s'articulent avec nos travaux et les étendent ; elles s'inscrivent sur le long terme dans la définition d'un environnement complet d'aide au développement de protocoles sûrs.

Index

Bibliography

[1] Jordan Adamek, Mikhail Nesterenko, and Sébastien Tixeuil. Evaluating and optimizing stabilizing dining philosophers. In *11th European Dependable Computing Conference, EDCC 2015, Paris, France, September 7-11, 2015*, pages 233–244. IEEE, 2015.

[2] Noa Agmon and David Peleg. Fault-tolerant gathering algorithms for autonomous mobile robots. *SIAM Journal of Computing*, 36(1):56–82, 2006.

[3] Karine Altisen, Pierre Corbineau, and Stéphane Devismes. Squeezing Streams and Composition of Self-stabilizing Algorithms. In Jorge A. Pérez and Nobuko Yoshida, editors, *39th International Conference on Formal Techniques for Distributed Objects, Components, and Systems (FORTE)*, volume LNCS-11535 of *Formal Techniques for Distributed Objects, Components, and Systems*, pages 21–38, Copenhagen, Denmark, June 2019. Springer International Publishing. Part 1: Full Papers.

[4] Hideki Ando, Ichiro Suzuki, and Masafumi Yamashita. Formation and agreement problems for synchronous mobile robots with limited visibility. In *Proceedings of Tenth International Symposium on Intelligent Control*, pages 453–460, January 1995. Proceedings of the 10th IEEE International Symposium on Intelligent Control ; Conference date: 27-08-1995 Through 29-08-1995.

[5] Philippe Audebaud and Christine Paulin-Mohring. Proofs of Randomized Algorithms in Coq. *Science of Computer Programming*, 74(8):568–589, 2009.

[6] Cédric Auger, Zohir Bouzid, Pierre Courtieu, Sébastien Tixeuil, and Xavier Urbain. Certified Impossibility Results for Byzantine-Tolerant Mobile Robots. In Teruo Higashino, Yoshiaki Katayama, Toshimitsu Masuzawa, Maria Potop-Butucaru, and Masafumi Yamashita, editors, *Stabilization, Safety, and Security of Distributed Systems - 15th International Symposium (SSS 2013)*, volume 8255 of *Lecture Notes in Computer Science*, pages 178–186, Osaka, Japan, November 2013. Springer-Verlag.

[7] Thibaut Balabonski, Pierre Courtieu, Robin Pelle, Lionel Rieg, Sébastien Tixeuil, and Xavier Urbain. Brief Announcement: Continuous vs. Discrete Asynchronous Moves: a Certified Approach for Mobile Robots. In Taisuke Izumi and Petr Kuznetsov, editors, *Stabilization, Safety, and Security of Distributed Systems - 20th International Symposium, (SSS 2018)*, volume 11201 of *Lecture Notes in Computer Science*, Tokyo, Japan, November 2018. Springer-Verlag.

[8] Thibaut Balabonski, Pierre Courtieu, Robin Pelle, Lionel Rieg, Sébastien Tixeuil, and Xavier Urbain. Continuous vs. discrete asynchronous moves: A certified approach for mobile robots. In Mohamed Faouzi Atig and Alexander A. Schwarzmann, editors, *Networked Systems - 7th International Conference, NETYS 2019, Marrakech, Morocco, June 19-21, 2019, Revised Selected Papers*, volume 11704 of *Lecture Notes in Computer Science*, pages 93–109. Springer-Verlag, 2019.

[9] Thibaut Balabonski, Pierre Courtieu, Robin Pelle, Lionel Rieg, Sébastien Tixeuil, and Xavier Urbain. Du discrètement continu au continûment discret. In *ALGOTEL 2020 – 22èmes Rencontres Francophones sur les Aspects Algorithmiques des Télécommunications*, Lyon, France, September 2020.

[10] Thibaut Balabonski, Pierre Courtieu, Robin Pelle, Lionel Rieg, Sébastien Tixeuil, and Xavier Urbain. Manuel de savoir-prouver à l'usage des roboteux et des distributeux. In *ALGOTEL 2019 - 21èmes Rencontres Francophones sur les Aspects Algorithmiques des Télécommunications*, pages 1–4, Saint Laurent de la Cabrerisse, France, 2019.

[11] Thibaut Balabonski, Amélie Delga, Lionel Rieg, Sébastien Tixeuil, and Xavier Urbain. Synchronous gathering without multiplicity detection: A certified algorithm. *Theory of Computing Systems*, 2018. https://doi.org/10.1007/s00224-017-9828-z.

[12] Thibaut Balabonski, Robin Pelle, Lionel Rieg, and Sébastien Tixeuil. A foundational framework for certified impossibility results with mobile robots on graphs. In Paolo Bellavista and Vijay K. Garg, editors, *Proceedings of the 19th International Conference on Distributed Computing and Networking, ICDCN 2018, Varanasi, India, January 4-7, 2018*, pages 5:1–5:10. ACM, 2018.

[13] Roberto Baldoni, François Bonnet, Alessia Milani, and Michel Raynal. On the solvability of anonymous partial grids exploration by mobile robots. In Theodore P. Baker, Alain Bui, and Sébastien Tixeuil, editors, *Principles of Distributed Systems*, pages 428–445, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[14] Eduardo Mesa Barrameda, Shantanu Das, and Nicola Santoro. Deployment of asynchronous robotic sensors in unknown orthogonal environments. In Sándor P. Fekete, editor, *Algorithmic Aspects of Wireless Sensor Networks*, pages 125–140, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[15] L. Barriere, P. Flocchini, E. Mesa-Barrameda, and N. Santoro. Uniform scattering of autonomous mobile robots in a grid. In *2009 IEEE International Symposium on Parallel Distributed Processing*, pages 1–8, 2009.

[16] Béatrice Bérard, Pierre Courtieu, Laure Millet, Maria Potop-Butucaru, Lionel Rieg, Nathalie Sznajder, Sébastien Tixeuil, and Xavier Urbain. Formal Methods for Mobile Robots: Current Results and Open Problems. *International Journal of Informatics Society*, 7(3):101–114, 2015. Invited Paper.

[17] Béatrice Berard, Laure Millet, Maria Potop-Butucaru, Yann Thierry-Mieg, and Sébastien Tixeuil. Formal verification of Mobile Robot Protocols. Technical report, LIP6 , LINCS , IUF, May 2013.

[18] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer-Verlag, 2004.

[19] S. Bhagat, S. Gan Chaudhuri, and K. Mukhopadhyaya. Fault-tolerant gathering of asynchronous oblivious mobile robots under one-axis agreement. *J. of Discrete Algorithms*, 36(C):50–62, January 2016.

[20] Lélia Blin, Alessia Milani, Maria Potop-Butucaru, and Sébastien Tixeuil. Exclusive perpetual ring exploration without chirality. In Nancy A. Lynch and Alexander A. Shvartsman, editors, *Distributed Computing, 24th International Symposium (DISC 2010)*, volume 6343 of *Lecture Notes in Computer Science*, pages 312–327, Cambridge, MA, USA, September 2010. Springer-Verlag.

[21] Zohir Bouzid, Shlomi Dolev, Maria Potop-Butucaru, and Sébastien Tixeuil. RoboCast: Asynchronous Communication in Robot Networks. In Chenyang Lu, Toshimitsu Masuzawa, and Mohamed Mosbah, editors, *OPODIS*, volume 6490 of *Lecture Notes in Computer Science*, pages 16–31, Tozeur, Tunisia, December 2010. Springer-Verlag.

[22] Quentin Bramas, Stéphane Devismes, and Pascal Lafourcade. Vers l'infini et au delà. In *ALGOTEL 2020 – 22èmes Rencontres Francophones sur les Aspects Algorithmiques des Télécommunications*, Lyon, France, September 2020.

[23] Quentin Bramas and Sébastien Tixeuil. The random bit complexity of mobile robots scattering. In Symeon Papavassiliou and Stefan Ruehrup, editors, *Ad-hoc, Mobile, and Wireless Networks - 14th International Conference, ADHOC-NOW 2015, Athens, Greece, June 29 - July 1, 2015, Proceedings*, volume 9143 of *Lecture Notes in Computer Science*, pages 210–224. Springer-Verlag, 2015.

[24] Davide Canepa and Maria Gradinariu Potop-Butucaru. Stabilizing flocking via leader election in robot networks. In Toshimitsu Masuzawa and Sébastien Tixeuil, editors, *Stabilization, Safety, and Security of Distributed Systems*, pages 52–66, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[25] Pierre Castéran and Vincent Filou. Tasks, types and tactics for local computation systems. *Studia Informatica Universalis*, 9(1):39–86, 2011.

[26] Reuven Cohen and David Peleg. Convergence Properties of the Gravitational Algorithm in Asynchronous Robot Systems. *SIAM Journal of Computing*, 34(6):1516–1528, 2005.

[27] Reuven Cohen and David Peleg. Local spreading algorithms for autonomous robot systems. *Theoretical Computer Science*, 399(1):71 – 82, 2008. Structural Information and Communication Complexity (SIROCCO 2006).

[28] Thierry Coquand and Christine Paulin-Mohring. Inductively Defined Types. In Per Martin-Löf and Grigori Mints, editors, *International Conference on Computer Logic (Colog'88)*, volume 417 of *Lecture Notes in Computer Science*, pages 50–66. Springer-Verlag, 1990.

[29] Andreas Cord-Landwehr, Bastian Degener, Matthias Fischer, Martina Hüllmann, Barbara Kempkes, Alexander Klaas, Peter Kling, Sven Kurras, Marcus Märtens, Friedhelm Meyer auf der Heide, Christoph Raupach, Kamil Swierkot, Daniel Warner, Christoph Weddemann, and Daniel Wonisch. Collisionless gathering of robots with an extent. In *SOFSEM 2011: Theory and Practice of Computer Science*, pages 178–189, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[30] Pierre Courtieu, Lionel Rieg, Sébastien Tixeuil, and Xavier Urbain. A Certified Universal Gathering Algorithm for Oblivious Mobile Robots. *CoRR*, abs/1506.01603, 2015.

[31] Pierre Courtieu, Lionel Rieg, Sébastien Tixeuil, and Xavier Urbain. Impossibility of Gathering, a Certification. *Information Processing Letters*, 115:447–452, 2015.

[32] Pierre Courtieu, Lionel Rieg, Sébastien Tixeuil, and Xavier Urbain. Certified universal gathering algorithm in $\mathbb{R}^2$ for oblivious mobile robots. In Cyril Gavoille and David Ilcinkas, editors, *Distributed Computing - 30th International Symposium, (DISC 2016)*, volume 9888 of *Lecture Notes in Computer Science*, Paris, France, September 2016. Springer-Verlag.

[33] Denis Cousineau, Damien Doligez, Leslie Lamport, Stephan Merz, Daniel Ricketts, and Hernán Vanzetto. TLA + Proofs. In Dimitra Giannakopoulou and Dominique Méry, editors, *FM*, volume 7436 of *Lecture Notes in Computer Science*, pages 147–154, Paris, France, August 2012. Springer-Verlag.

[34] Jurek Czyzowicz, Leszek Gąsieniec, and Andrzej Pelc. Gathering few fat mobile robots in the plane. *Theoretical Computer Science*, 410(6):481 – 499, 2009. Principles of Distributed Systems.

[35] S. Das, P. Flocchini, G. Prencipe, N. Santoro, and M. Yamashita. The power of lights: Synchronizing asynchronous robots using visible bits. In *2012 IEEE 32nd International Conference on Distributed Computing Systems*, pages 506–515, 2012.

[36] Stéphane Devismes. Optimal exploration of small rings. In *Proceedings of the Third International Workshop on Reliability, Availability, and Security*, WRAS '10, New York, NY, USA, 2010. Association for Computing Machinery.

[37] Stéphane Devismes, Franck Petit, and Sébastien Tixeuil. Optimal probabilistic ring exploration by semi-synchronous oblivious robots. In Shay Kutten and Janez Zerovnik, editors, *Structural Information and Communication Complexity, 16th International Colloquium, SIROCCO 2009, Piran, Slovenia, May 25-27, 2009, Revised Selected Papers*, volume 5869 of *Lecture Notes in Computer Science*, pages 195–208. Springer, 2009.

[38] Yoann Dieudonné and Franck Petit. Scatter of weak robots. *Parallel Processing Letters*, 19, 02 2007.

[39] Ha Thi Thu Doan, François Bonnet, and Kazuhiro Ogata. Model checking of robot gathering. In James Aspnes, Alysson Bessani, Pascal Felber, and João Leitão, editors, *21st International Conference on Principles of Distributed Systems, OPODIS 2017, Lisbon, Portugal, December 18-20, 2017*, volume 95 of *LIPIcs*, pages 12:1–12:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017.

[40] Swan Dubois and Sébastien Tixeuil. A Taxonomy of Daemons in Self-stabilization. Technical Report 1110.0334, ArXiv eprint, October 2011.

[41] Xavier Défago. Model-Checking for Gathering with lights in Euclidian spaces. In *Mobile Robots Verification (MoRoVer'17)*, Lyon, France, November 2017. https://morover.sciencesconf.org.

[42] Xavier Défago and Samia Souissi. Non-uniform circle formation algorithm for oblivious mobile robots with convergence toward uniformity. *Theoretical Computer Science*, 396(1-3):97–112, 2008.

[43] Paola Flocchini, David Ilcinkas, Andrzej Pelc, and Nicola Santoro. Remembering without memory: Tree exploration by asynchronous oblivious robots. *Theoretical Computer Science*, 411(14-15):1583–1598, 2010.

[44] Paola Flocchini, David Ilcinkas, Andrzej Pelc, and Nicola Santoro. Computing without communicating: Ring exploration by asynchronous oblivious robots. *Algorithmica*, 65(3):562–583, 2013.

[45] Paola Flocchini, Giuseppe Prencipe, and Nicola Santoro. Self-deployment algorithms for mobile sensors on a ring. In Sotiris E. Nikoletseas and José D. P. Rolim, editors, *Algorithmic Aspects of Wireless Sensor Networks*, pages 59–70, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[46] Paola Flocchini, Giuseppe Prencipe, Nicola Santoro, and Peter Widmayer. Hard tasks for weak robots: The role of common knowledge in pattern formation by autonomous mobile robots. In Alok Aggarwal and C. Pandu Rangan, editors, *Algorithms and Computation, 10th International Symposium, ISAAC '99, Chennai, India, December 16-18, 1999, Proceedings*, volume 1741 of *Lecture Notes in Computer Science*, pages 93–102. Springer, 1999.

[47] Sruti Gan Chaudhuri and Krishnendu Mukhopadhyaya. Gathering asynchronous transparent fat robots. In Tomasz Janowski and Hrushikesha Mohanty, editors, *Distributed Computing and Internet Technology*, pages 170–175, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[48] Georges Gonthier. *The Four Colour Theorem: Engineering of a Formal Proof*, page 333. Springer-Verlag, Berlin, Heidelberg, 2008.

[49] Georges Gonthier. Engineering Mathematics: the Odd Order Theorem Proof. In Roberto Giacobazzi and Radhia Cousot, editors, *POPL*, pages 1–2. ACM, 2013.

[50] John Harrison. Hol light: An overview. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *Theorem Proving in Higher Order Logics*, pages 60–66, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[51] William A. Howard. The formulae-as-types notion of construction. In J. Roger Hindley Jonathan P. Seldin, editor, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 479–490. Academic Press, London, 1980.

[52] Taisuke Izumi, Yoshiaki Katayama, Nobuhiro Inuzuka, and Koichi Wada. Gathering autonomous mobile robots with dynamic compasses: An optimal result. In Andrzej Pelc, editor, *Distributed Computing*, pages 298–312, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[53] Taisuke Izumi, Maria Gradinariu Potop-Butucaru, and Sébastien Tixeuil. Connectivity-preserving scattering of mobile robots with limited visibility. In Shlomi Dolev, Jorge Cobb, Michael Fischer, and Moti Yung, editors, *Stabilization, Safety, and Security of Distributed Systems*, pages 319–331, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[54] Tomoko Izumi, Taisuke Izumi, Sayaka Kamei, and Fukuhito Ooshita. Mobile robots gathering algorithm with local weak multiplicity in rings. In Boaz Patt-Shamir and Tinaz Ekim, editors, *Structural Information and Communication Complexity, 17th International Colloquium, SIROCCO 2010, Sirince, Turkey, June 7-11, 2010. Proceedings*, volume 6058 of *Lecture Notes in Computer Science*, pages 101–113. Springer-Verlag, 2010.

[55] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. Sel4: Formal verification of an os kernel. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, page 207–220, New York, NY, USA, 2009. Association for Computing Machinery.

[56] Anissa Lamani, Maria Gradinariu Potop-Butucaru, and Sébastien Tixeuil. Optimal deterministic ring exploration with oblivious asynchronous robots. In Boaz Patt-Shamir and Tinaz Ekim, editors, *Structural Information and Communication Complexity, 17th International Colloquium, SIROCCO 2010, Sirince, Turkey, June 7-11, 2010. Proceedings*, volume 6058 of *Lecture Notes in Computer Science*, pages 183–196. Springer-Verlag, 2010.

[57] Leslie Lamport. The temporal logic of actions. *ACM Trans. Program. Lang. Syst.*, 16(3):872–923, May 1994.

[58] Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine Generals Problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, 1982.

[59] Thierry Lecomte. Safe and reliable metro platform screen doors control/command systems. In Jorge Cuellar, Tom Maibaum, and Kaisa Sere, editors, *FM 2008: Formal Methods*, pages 430–434, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[60] Xavier Leroy. A Formally Verified Compiler Back-End. *Journal of Automated Reasoning*, 43(4):363–446, 2009.

[61] Maja J. Mataric. *Interaction and Intelligent Behavior*. PhD thesis, USA, 1995. Not available for Univ. Microfilms Int.

[62] William McCune. Solution of the robbins problem. *J. Autom. Reason.*, 19(3):263–276, December 1997.

[63] Robin Milner. *Communicating and mobile systems: the pi calculus*. Cambridge university press, 1999.

[64] César A. Muñoz, Gilles Dowek, and Víctor Carreño. Modeling and verification of an air traffic concept of operations. *SIGSOFT Softw. Eng. Notes*, 29(4):175–182, July 2004.

[65] Iñaki Navarro, Álvaro Gutiérrez, Fernando Matía, and Félix Monasterio-Huelin. An approach to flocking of robots using minimal local sensing and common orientation. In Emilio Corchado, Ajith Abraham, and Witold Pedrycz, editors, *Hybrid Artificial Intelligence Systems*, pages 616–624, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[66] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002.

[67] Fukuhito Ooshita and Sébastien Tixeuil. On the self-stabilization of mobile oblivious robots in uniform rings. In Andréa W. Richa and Christian Scheideler, editors, *Stabilization, Safety, and Security of Distributed Systems*, pages 49–63, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[68] Maria Potop-Butucaru, Nathalie Sznajder, Sébastien Tixeuil, and Xavier Urbain. Formal methods for mobile robots. In Paola Flocchini, Giuseppe Prencipe, and Nicola Santoro, editors, *Distributed Computing by Mobile Entities*, volume 11340 of *Lecture Notes in Computer Science, Theoretical Computer Science and General Issues*, pages 278–313. Springer Nature, 2019.

[69] Giuseppe Prencipe. Impossibility of gathering by a set of autonomous mobile robots. *Theoretical Computer Science*, 384(2-3):222–231, 2007.

[70] Laurent Reynaud. *Stratégies de mobilité optimisées pour la tolérance aux perturbations dans les réseaux sans fil*. PhD thesis, Université Claude Bernard (Lyon), 2017. Thèse de doctorat dirigée par Guérin-Lassous, Isabelle Informatique Lyon 2017.

[71] Craig W. Reynolds. Flocks, herds and schools: A distributed behavioral model. In *Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '87, page 25–34, New York, NY, USA, 1987. Association for Computing Machinery.

[72] Samia Souissi, Yan Yang, and Xavier Défago. Fault-tolerant flocking in a k-bounded asynchronous system. In Theodore P. Baker, Alain Bui, and Sébastien Tixeuil, editors, *Principles of Distributed Systems*, pages 145–163, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[73] Kazuo Sugihara and Ichiro Suzuki. Distributed algorithms for formation of geometric patterns with many mobile robots. *Journal of Robotic Systems*, 13(3):127–139, 1996.

[74] Ichiro Suzuki and Masafumi Yamashita. Distributed Anonymous Mobile Robots: Formation of Geometric Patterns. *SIAM Journal of Computing*, 28(4):1347–1363, 1999.

[75] Nikhil Swamy, Juan Chen, and Ben Livshits. Verifying higher-order programs with the dijkstra monad. In *ACM Programming Language Design and Implementation (PLDI) 2013*. ACM, June 2013.

[76] Josef Urban and Geoff Sutcliffe. Automated reasoning and presentation support for formalizing mathematics in mizar. *CoRR*, abs/1005.4592, 2010.

[77] P. K. C. Wang. Navigation strategies for multiple autonomous mobile robots moving in formation. *Journal of Robotic Systems*, 8(2):177–195, 1991.

[78] Naixue Xiong, He Jing, Yang Yan, He Yanxiang, Kim Tai-hoon, and Lin Chuan. A survey on decentralized flocking schemes for a set of autonomous mobile robots (invited paper). *Journal of Communications*, 5, 01 2010.

[79] Yan Yang, Samia Souissi, Xavier Défago, and Makoto Takizawa. Fault-tolerant flocking for a group of autonomous mobile robots. *Journal of Systems and Software*, 84(1):29 – 36, 2011. Information Networking and Software Services.

[80] Daisuke Yoshida, Toshimitsu Masuzawa, and Hideo Fujiwara. Fault-tolerant distributed algorithms for autonomous mobile robots with crash faults. *Systems and Computers in Japan*, 28(2):33–43, 1997.

université PARIS-SACLAY

ÉCOLE DOCTORALE
Sciences et technologies de l'information et de la communication (STIC)

Titre: Contributions à la modélisation formelle d'essaims de robots mobiles

Mots clés: Preuve formelle, Algorithmique distribuée, Robots mobiles autonomes.

Résumé: L'algorithmique distribuée fait partie des domaines où le raisonnement informel n'est pas une option, en particulier lorsque des erreurs dites byzantines peuvent survenir. Elle est également caractérisée par une grande diversité de modèles dont les modulations subtiles impliquent des propriétés radicalement différentes. Nous nous intéressons aux « réseaux de robots » : nuages d'entités autonomes devant accomplir une tâche en coopération. Les applications que laissent envisager ces essaims d'agents sont extrêmement prometteuses : exploration et recherche de survivants dans des zones dévastées, patrouilles et vols de drones en formation, etc. Ces quelques exemples potentiellement critiques soulignent la grande dynamicité du modèle; ils indiquent également à quel point des défaillances des robots ou des erreurs dans les protocoles distribués qui les équipent peuvent avoir de désastreuses conséquences. Pour garantir la sûreté des protocoles et la sécurité des tâches, nous visons à l'obtention, à l'aide de l'assistant à la preuve Coq, de validations mécaniques formelles de propriétés de certains protocoles distribués. Un prototype de modèle formel Coq pour les réseaux de robots, Pactole, a récemment montré la faisabilité d'une approche de vérification par assistant à la preuve dans ce cadre. Il capture assez naturellement de nombreuses variantes de ces réseaux, notamment en ce qui concerne la topologie ou les propriétés des démons. Ce modèle est bien sûr à l'ordre supérieur et s'appuie sur des types coinductifs. Il permet de démontrer en Coq à la fois des propriétés positives : le programme embarqué permet de réaliser la tâche quelle que soit la configuration de départ, comme des propriétés négatives : il n'existe aucun programme embarqué permettant de réaliser la tâche. Dans le cadre émergent des réseaux de robots, les modèles sont distingués par les caractéristiques et capacités des robots, la topologie de l'espace dans lequel ils évoluent, le degré de synchronisme (modélisé par les propriétés du démon d'activation), les erreurs pouvant survenir, etc. Le prototype Pactole n'exprime que certaines de ces variantes. Pensé dans un cadre théorique (robots ponctuels, déplacements instantanés, etc.), des hypothèses restent hors de sa portée, en particulier des hypothèses réalistes comme des exécutions totalement asynchrones ou des risques de collision. L'absence de collision est fondamentale dans toutes les applications liées aux évolutions en formation (drones) et une condition de sécurité critique dès qu'on s'intéresse au transport aérien. Une validation formelle de cette propriété revêt donc une grande importance. Le travail consiste à étendre le modèle formel afin de prendre en compte des évolutions asynchrones de robots volumineux. Cette modélisation doit permettre une formulation aisée de protocoles et des tâches qu'ils sont censés réaliser. On s'intéressera en particulier à garantir l'absence de collision au cours de déplacements potentiellement complexes.

Title: Contribution to the formal modelling of mobile robot swarms

Keywords: Formal Proof, Distributed algorithms, Autonomous mobile robots

Abstract: Distributed Algorithm is among domains where informal reasoning is not an option, especially when Byzantine errors may occur. It is also characterized by a large variety of models whose subtle modulations imply radically different properties. We are interested in "robotic network": clouds of autonomous entities performing a cooperative task. The applications that these swarms of agents offer are extremely promising: exploration and search for survivors in devasted areas, patrols and drone flights in formation, etc. These few potentially critical exemples underline the high dynamicity of the model; they also indicate how failures of robots or errors in the distributed protocols that equip them can have distastrous consequences. To ensure the safety of the protocols and the security of tasks, we aim to optain, with the help of the Coq proof assistant, foraml mechanical validations of properties of certain distributed protocols. A prototype of Coq's formal model for robot network, Pactole, has recently shown the feasibility of an proof assistant verification in this framework. It captures quite naturally many variants of these networks, especially with regard to topology or the properties of demons. This model is of course in the higher order, and is based on coinductive types. It makes it possible to demonstrate in Coq both positive properties: the embedded program makes it possible to carry out the task regardless of the initial configuration, as negative properties: there is no embedded program to complete the task. In the emerging framework of robot networks, the models are distinguished by the characteristics and capabilities of the robots, the topology of the space in which they evolve, the degree of synchronism (modelled by the properties of the activation demon), the error that can occure, etc. The prototype Pactole expresses only some of these variants. Thought in a theoretical framework (point robots, instantaneous movement, etc.), hypotheses remain out of reach, in particular realistic hypothesis such as totally asynchronous executions, or risk of collision. The absence of collision is fundamental in all applications relates to formation flights (drones) and critical safety condition as soon as one is interested in air transport. Formal validation of this property is therefore of great importance. The work consits of extending the formal model to take into account asynchronous evolutions of large robots. This modelling should allow easy formulation of protocols and the task they are supposed to perform. Particular attention will be given to ensuring the absence of collisions during potentially complex movements.

Université Paris-Saclay
Espace Technologique / Immeuble Discovery
Route de l'Orme aux Merisiers RD 128 / 91190 Saint-Aubin, France